UNIVERSIDADE FEDERAL DE SANTA CATARINA
CAMPUS REITOR JOÃO DAVID FERREIRA LIMA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Henrique dos Santos Goulart

UMA ESTRATÉGIA DE REBALANCEAMENTO DE ESTADOS PARA CHECKPOINTS PARTICIONADOS

Florianópolis, Santa Catarina – Brasil
2023

Henrique dos Santos Goulart

UMA ESTRATÉGIA DE REBALANCEAMENTO DE ESTADOS PARA CHECKPOINTS PARTICIONADOS

Dissertação submetida ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Santa Catarina para a obtenção do Grau de Mestre em Ciência da Computação.
Orientador: Odorico Machado Mendizabal, Dr.

Florianópolis, Santa Catarina – Brasil
2023

Ficha de identificação da obra elaborada pelo autor,
através do Programa de Geração Automática da Biblioteca Universitária da UFSC.

Goulart, Henrique dos Santos
Uma estratégia de rebalanceamento de estados para checkpoints particionados / Henrique dos Santos Goulart ; orientador, Odorico Machado Mendizabal, 2023.
63 p.

Dissertação (mestrado) - Universidade Federal de Santa Catarina, Centro Tecnológico, Programa de Pós-Graduação em Ciência da Computação, Florianópolis, 2023.

Inclui referências.

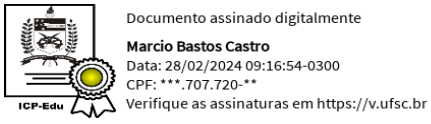
1. Ciência da Computação. 2. Checkpoint. 3. Reparticionamento. 4. Balanceamento de carga. I. Mendizabal, Odorico Machado. II. Universidade Federal de Santa Catarina. Programa de Pós-Graduação em Ciência da Computação. III. Título.

Henrique dos Santos Goulart

UMA ESTRATÉGIA DE REBALANCEAMENTO DE ESTADOS PARA CHECKPOINTS PARTICIONADOS

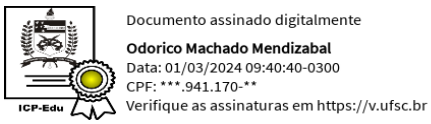
Esta dissertação foi julgada adequada para obtenção do Título de Mestre em Ciência da Computação, e foi aprovado em sua forma final pelo Programa de Pós-Graduação em Ciência da Computação do INE – Departamento de Informática e Estatística, CTC – Centro Tecnológico da Universidade Federal de Santa Catarina.

Florianópolis, Santa Catarina – Brasil, 8 de dezembro de 2023.



Márcio Bastos Castro, Dr.
Coordenador do Programa de Pós-Graduação em Ciência da Computação

Banca Examinadora:



Odorico Machado Mendizabal, Dr.
Orientador
Universidade Federal de Santa Catarina – UFSC

Prof. Luciana de Oliveria Rech, Dra.
Univesidade Federal de Santa Catarina – UFSC

Prof. Santiago Valdés Ravelo, Dr.
Universidade Estadual de Campinas – UNICAMP

Prof. Eduardo Camilo Inacio, Dr.
UniSENAI

AGRADECIMENTOS

Em primeiro lugar, expresso minha profunda gratidão ao meu orientador, Dr. Odorico Mendizabal, cujo apoio e orientação foram essenciais durante todo o curso de mestrado. Seu conhecimento e dedicação foram fundamentais para a concretização deste trabalho.

Agradeço também ao professor Dr. Álvaro Franco. Sua participação nas discussões, contribuições com ideias e, em especial, na formulação matemática do problema, foi imprescindível. Juntamente com o Dr. Prof. Odorico, o professor Álvaro foi uma peça-chave no desenvolvimento desta pesquisa e nos artigos relacionados.

Meus agradecimentos estendem-se à minha família, meu pai Everaldo Goulart, minha mãe Eliane Vieira dos Santos e meu irmão Vinícius dos S. Goulart que estiveram ao meu lado oferecendo apoio. Também quero agradecer a minha noiva, Ana Carolina F. da Silva, cuja paciência e compreensão foram essenciais durante esse período.

Agradeço ao Programa de Pós-Graduação em Ciência da Computação (PPGCC) da Universidade Federal de Santa Catarina (UFSC) por fornecer a oportunidade e o ambiente propício para o desenvolvimento acadêmico e profissional.

Por fim, mas não menos importante, reconheço os esforços e a persistência que dediquei a este trabalho. Esta pesquisa é também um testemunho da minha determinação e compromisso com a excelência acadêmica.

RESUMO

Esta pesquisa aborda os métodos de *checkpoint/recuperação*, que registram periodicamente o estado do sistema em momentos sem falhas, permitindo a recuperação a partir de um estado anterior estável. Embora o *checkpoint* acarrete custos, demandando um equilíbrio entre o armazenamento das imagens do estado do sistema e o gerenciamento de solicitações recebidas, uma abordagem promissora para agilizar o *checkpoint* é a divisão do estado do serviço, possibilitando o salvamento simultâneo de partições individuais. Esta estratégia não só potencializa o desempenho através do aumento do paralelismo no processamento das solicitações, mas também, como demonstrado pelos resultados desta pesquisa, permite um melhor rebalanceamento de carga e otimização da vazão sem custos adicionais. No entanto, a pesquisa revela que, apesar destes avanços, houve apenas uma modesta melhoria na eficiência do processo de *checkpoint* em si. A pesquisa propõe uma metodologia que combina *checkpoint* paralelo com uma técnica de repartição flexível baseada em grafos, definindo formalmente o problema e realizando uma análise detalhada do desempenho da abordagem proposta. Experimentos práticos demonstram as vantagens dos *checkpoints* paralelos e ressaltam as melhorias de eficiência alcançadas com a repartição baseada em grafos. Além disso, ao comparar um método de particionamento *round-robin* com uma abordagem dinâmica, o estudo enfoca o grau de paralelismo atingido pelas threads de *checkpoint* e o impacto das diferentes estratégias de repartição na eficiência do *checkpoint* e no desempenho geral da aplicação. Os resultados confirmam que é possível alcançar um desempenho superior no sistema para operações que envolvem o acesso a múltiplas partições e exigem coordenação entre elas, sem incorrer em custos adicionais para o processo de *checkpoint*. Assim, a pesquisa fornece percepções importantes sobre como a otimização de processos em sistemas distribuídos pode ser realizada sem comprometer a eficiência ou acarretar custos adicionais.

Keywords: checkpoint/recuperação; balanceamento de carga; reparticionamento;

ABSTRACT

This research addresses *checkpoint/recovery* methods, which periodically record the system's state during fault-free moments, enabling recovery from a previously stable state. Although the *checkpoint* incurs costs, requiring a balance between storing the system state images and managing received requests, a promising approach to expedite the *checkpoint* is the division of the service state, allowing for the simultaneous saving of individual partitions. This strategy not only enhances performance through increased parallelism in request processing but also, as demonstrated by the results of this research, allows for improved load rebalancing and throughput optimization without additional costs. However, the research reveals that, despite these advancements, there was only a modest improvement in the efficiency of the *checkpoint* process itself. The research proposes a methodology that combines parallel *checkpoint* with a flexible graph-based partitioning technique, formally defining the problem and conducting a detailed analysis of the performance of the proposed approach. Practical experiments demonstrate the advantages of parallel *checkpoints* and highlight the efficiency improvements achieved with graph-based partitioning. Furthermore, by comparing a *round-robin* partitioning method with a dynamic approach, the study focuses on the degree of parallelism achieved by the *checkpoint* threads and the impact of different partitioning strategies on the *checkpoint*'s efficiency and the overall performance of the application. The results confirm that it is possible to achieve superior system performance for operations that involve accessing multiple partitions and require coordination among them, without incurring additional costs for the *checkpoint* process. Thus, the research provides important insights into how process optimization in distributed systems can be accomplished without compromising efficiency or incurring additional costs.

Keywords: checkpoint/restore; load balancing; repartitioning;

LISTA DE FIGURAS

LISTA DE ABREVIATURAS E SIGLAS

CRIU	*Checkpoint/Restore In Userspace*
E/S	Entrada/Saída
NVM	*Non-Volatile Memory*
BLCR	*Berkeley Lab Checkpoint Recovery(BLCR)*
CRAK	*Checkpoint/Restart As a Kernel Module*
HEM	*Heavy Edge Matching*
YCSB	*Yahoo! Cloud Serving Benchmark*
GB	Gigabyte
kB	Kilobyte
P-SMR	*Parallel State-Machine Replication*

SUMÁRIO

1 INTRODUÇÃO

Em um mundo cada vez mais dependente de sistemas de alta disponibilidade, a necessidade de manter serviços ininterruptos, mesmo diante de falhas inesperadas ou ameaças potenciais, torna-se uma prioridade. Nesse contexto, sistemas tolerantes a falhas se destacam como uma técnica essencial. Esses sistemas podem usar a abordagem de *checkpoint/recovery*, combinando estratégias de durabilidade para garantir a robustez e a acessibilidade contínua do sistema.

O funcionamento dessa técnica consiste na captura periódica de estado do sistema, permitindo a recuperação do sistema na ocorrência de falhas e a restauração deste para estados não muito defasados em relação ao restante do ambiente (ELNOZAHY; ALVISI *et al.*, 2002; EGWUTUOHA *et al.*, 2013). No entanto, a criação desses *checkpoints* não é isenta de desafios, pois requer uma coordenação com as requisições em andamento. À medida que o sistema evolui e cresce em complexidade, as operações intensivas de E/S envolvidas na criação de *checkpoints* podem se tornar um gargalo crítico.

Uma resposta ao desafio de sincronizar as requisições em andamento surge por meio do particionamento do estado do sistema. Essa abordagem, especialmente relevante em arquiteturas com multi-processadores, permite que partes distintas do estado do sistema sejam tratadas independentemente. Dessa forma, as partições do estado podem ser salvas em paralelo, acelerando significativamente o processo de *checkpoint* (MENDIZABAL; DOTTI; PEDONE, 2016; BESSANI *et al.*, 2013). Além disso, a introdução de tecnologias de hardware específicas e suporte em nível de sistema para paralelismo, como E/S (Entrada/Saída) paralela (BOITO *et al.*, 2018), NVM (*Non-Volatile Memory*) (LEE *et al.*, 2019), e memória transacional (NAKANO *et al.*, 2006), pode contribuir para reduzir o sobrecusto com operações de *checkpoint* relacionadas à durabilidade.

Mais do que apenas aumentar o processamento de *checkpointing*, a técnica de particionamento do estado tem potencial para otimizar a vazão do sistema. Ao fragmentar o estado da aplicação em unidades menores e gerenciáveis, essa abordagem possibilita a paralelização eficaz do trabalho, resultando em um desempenho superior em comparação com o processamento sequencial. No entanto, determinar um particionamento eficiente do estado do serviço é uma tarefa complexa, que exige informações detalhadas sobre a carga de trabalho (TROMBETA; MENDIZABAL, 2020; BULUÇ *et al.*, 2016; TRIFUNOVIĆ; KNOTTENBELT, 2008; KARYPIS; KUMAR, 1998). Mesmo quando essas informações estão disponíveis, é importante estar ciente de que os padrões de acesso aos dados podem evoluir com o tempo, potencialmente gerando desequilíbrios e desafios adicionais (QUAMAR; KUMAR; DESHPANDE, 2013; GOULART; TROMBETA *et al.*, 2023).

1.1 PROBLEMA DE PESQUISA

Nesse contexto, surge uma questão intrigante: será possível realizar o reparticionamento e o rebalanceamento das partições durante a execução de *checkpoints*, sem afetar drasticamente o tempo necessário para sua conclusão? É sabido que o *checkpoint* exige uma quantidade considerável de E/S, enquanto a carga de processamento associada é relativamente menor (BESSANI *et al.*, 2013). Portanto, seria viável aproveitar esse momento de intenso uso de E/S para otimizar a distribuição das partições e alcançar um equilíbrio mais eficiente?

A presente pesquisa busca responder a essa pergunta, explorando a possibilidade de executar operações de reparticionamento e rebalanceamento durante a criação de *checkpoints*. Nosso objetivo é avaliar se é viável aprimorar a distribuição das partições do estado do sistema sem prejudicar substancialmente o desempenho do processo de *checkpoint*. Para tal, consideraremos a utilização de técnicas específicas de reparticionamento durante o processo de *checkpoint*, visando alcançar uma melhor distribuição de carga e uma maior eficiência no uso dos recursos do sistema.

Ao abordar essa questão, esperamos contribuir para o desenvolvimento de estratégias mais eficazes de manutenção da alta disponibilidade em sistemas tolerantes a falhas, oferecendo resultados e experimentos sobre como reduzir os custos de *checkpoint* e, por conseguinte, melhorar a robustez e a continuidade dos serviços em ambientes críticos.

1.2 OBJETIVOS

A pesquisa tem como objetivo principal investigar a viabilidade de realizar reparticionamento e rebalanceamento das partições durante a execução de *checkpoints* em sistemas resilientes a falhas. O foco central é determinar se essa abordagem pode ser aplicada de forma eficaz, otimizando a distribuição de carga e o equilíbrio das partições, sem afetar drasticamente o tempo necessário para a conclusão do processo de *checkpoint*.

A pesquisa busca, assim, contribuir para o avanço do conhecimento em sistemas tolerantes a falhas, oferecendo uma percepção valiosa sobre a melhor utilização de recursos durante a operação de *checkpoint*.

1.2.1 Objetivos Específicos

1. Investigar as operações de *checkpoint/recovery* em sistemas tolerantes a falhas e analisar as características que influenciam o desempenho dessas operações;

2. Formalizar o problema de otimização do reparticionamento de estados em sistemas particionados, utilizando grafos, com o objetivo de aprimorar a eficiência da

vazão e do processamento de *checkpoint*;

3. Avaliar o impacto do reparticionamento de partições do estado do sistema durante a execução de *checkpoints*, considerando a distribuição de carga de trabalho e o equilíbrio entre as partições;
4. Implementar um algoritmo e técnicas de reparticionamento de forma eficiente durante o processo de *checkpoint*;
5. Realizar experimentos para quantificar o desempenho e a eficácia das estratégias de reparticionamento propostas durante a criação de *checkpoints*;
6. Comparar o desempenho das técnicas de reparticionamento com a execução de *checkpoints* convencionais;
7. Propor recomendações e diretrizes práticas para a aplicação de reparticionamento durante *checkpoints*, com o objetivo de fornecer orientações sobre as circunstâncias ideais para sua implementação.

1.2.2 Organização do texto

Este trabalho organiza-se da seguinte maneira: o Capítulo 2 detalha e explora diferentes estratégias de *checkpoint*. No Capítulo 4, o funcionamento do estado particionado e *checkpoints* particionados é elucidado, definindo-se também o problema e a solução proposta por esta pesquisa. O Capítulo 5 apresenta o protótipo implementado, bem como a execução e análise dos resultados experimentais. Uma revisão abrangente da literatura é conduzida no Capítulo 3. Finalmente, o Capítulo 6 discute as conclusões derivadas desta investigação.

2 *CHECKPOINTS*: VARIEDADES E CONTEXTOS DE APLICAÇÃO

Neste capítulo, apresentaremos os fundamentos relacionados à técnica de *checkpoint* em sistemas tolerantes a falhas. Para compreender o funcionamento desses *checkpoints* e sua aplicação em ambientes críticos, abordaremos esse assunto em subseções específicas.

Começaremos explorando os diferentes tipos de *checkpoints*, suas características e suas aplicações em sistemas coordenados e não coordenados. Em seguida, discutiremos as variantes incrementais, particionadas, difusas, dinâmicas e adaptáveis dos *checkpoints*. Além disso, examinaremos como os *checkpoints* podem ser implementados em níveis de usuário e nível de sistema.

A análise subsequente se concentrará no Modelo de Execução de Sistemas com Estado Particionado. Será examinado o funcionamento do particionamento e a importância da sincronização na execução de operações em diferentes partições, especialmente em operações do tipo *swap/scan*. Além disso, será discutido o custo elevado associado à execução adequada dessas operações.

2.1 *CHECKPOINT*

Em sistemas de computação, particularmente em ambientes que demandam alta disponibilidade e consistência, a capacidade de recuperação após falhas é uma característica comumente necessária. Interrupções inesperadas do sistema, seja por falhas de software, falhas de hardware ou outros problemas, podem acarretar em perda de dados e tempo, afetando a integridade do sistema. Nesse contexto, os *checkpoints* atuam como uma ferramenta importante, auxiliando na mitigação desses riscos. Eles atuam como uma *fotografia* do estado do sistema em determinados momentos, garantindo que, em caso de falhas, o sistema possa ser restaurado a um estado conhecido e seguro. No entanto, enquanto os *checkpoints* oferecem essa segurança e robustez, eles contêm suas próprias implicações e desafios.

A Figura 1 ilustra a progressão temporal de operações em um sistema representado por um cliente e um processo p1. Ao longo do tempo, o cliente realiza várias operações que alteram o estado do processo p1. A linha do tempo, representada pelo eixo horizontal rotulado como *tempo* mostra a progressão das operações do cliente. O cliente realiza operações específicas que são refletidas no processo p1. Essas operações são:

1. set x=1: Define o valor de x como 1.
2. set y=34: Define o valor de y como 34.
3. get x: Solicita o valor de x.

4. set z=9: Define o valor de z como 9.

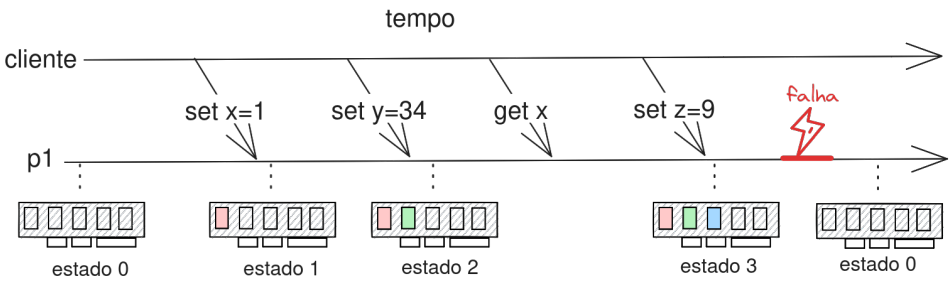


Figura 1 – Falha de um processo

O processo p1 mantém seu estado ao longo do tempo, que é alterado pelas operações do cliente. Dessa forma, os estados do processo sofrem as seguintes modificações:

- Estado 0: Estado inicial do processo.
- Estado 1: Estado após set x=1 ter sido executado.
- Estado 2: Estado após set y=34 ter sido executado.
- Estado 3: Estado após set z=9 ter sido executado.

Cada alteração de estado é representada por uma barra colorida dentro do processo p1, indicando que ocorreu uma mudança da informação armazenada na memória. Depois de set z=9, ocorre uma falha no sistema, representada pelo ícone de relâmpago. Após a falha, o sistema recorre ao último estado seguro, que neste caso é *Estado 0*. Isso indica que não foi efetuado nenhuma operação de *checkpoint* antes de uma falha ocorrer, assim, não há uma *foto* mais atual dos dados para serem recuperados.

Por outro lado, na Figura 2, observamos um *checkpoint* em ação, no qual os dados da memória são salvos em um disco rígido. Dessa forma, se ocorrer uma falha, é possível utilizar o arquivo armazenado para recuperar esses dados e restaurar o processo a um estado mais recente, em vez de retornar ao estado *Estado 0*.

A abordagem de recuperação fundamentada em *checkpoints* aprimora a execução dos sistemas ao possibilitar a persistência de comandos executados, viabilizando a retomada da execução do sistema após uma falha subsequente ao último *checkpoint* estabelecido. Entretanto, esses métodos apresentam efeitos adversos no desempenho do sistema em questão (BESSANI *et al.*, 2013; ZHENG *et al.*, 2014; MENDIZABAL; DOTTI; PEDONE, 2016). Por exemplo, as gravações síncronas referentes aos *checkpoints* podem ocasionar uma diminuição expressiva na vazão do sistema durante sua execução.

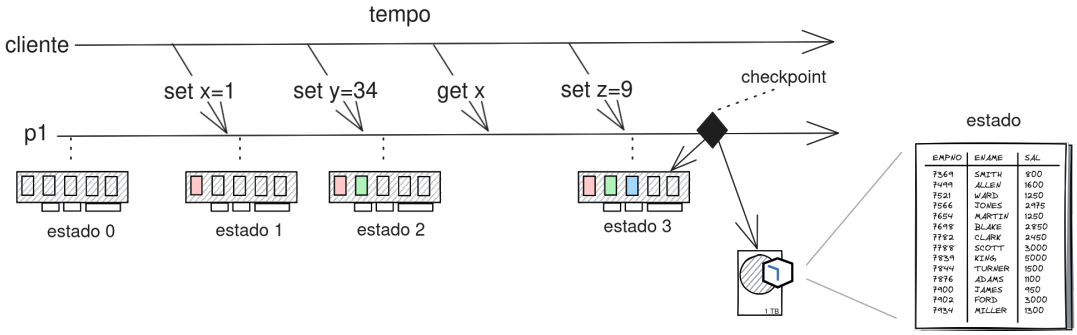


Figura 2 – *foto* durante o *checkpoint*

Um exemplo é ilustrado na Figura 3. A linha azul demonstra o número de requisições por segundo que o processo 1 (p1) atende ao longo do tempo. Inicialmente, o processo exibe uma vazão quase constante, oscilando levemente em torno de 250 req/s. Contudo, uma queda notável na vazão é observada ao se iniciar um *checkpoint*. Esta redução acentuada é consequência da operação de *checkpoint*, quando o sistema está salvando o estado atual do processo em um meio de armazenamento persistente, como um disco rígido, para a posterior recuperação em caso de falhas. Durante este período, as requisições não são executadas mas são acumuladas, aguardando para serem processadas após a conclusão do *checkpoint*.

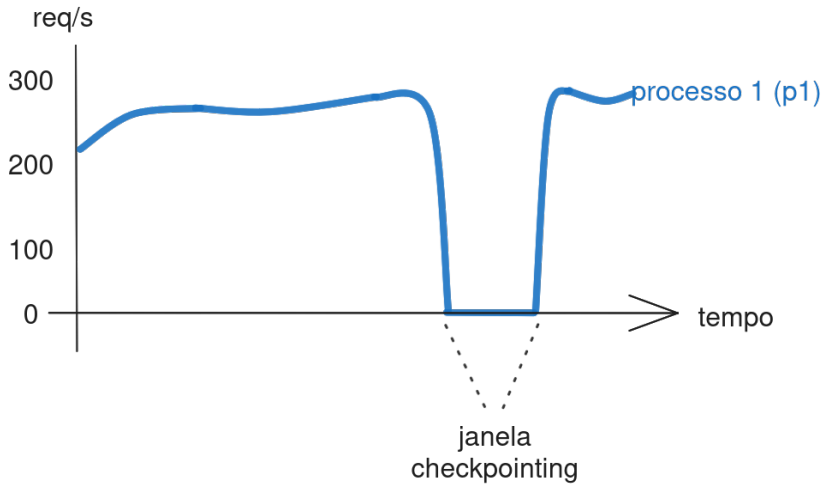


Figura 3 – *Checkpoint* em execução afetando operações

Durante a janela de *checkpoint*, é esperado que haja uma interrupção ou diminuição significativa no desempenho do sistema, uma vez que recursos como processador, memória e E/S são alocados para garantir que o estado do processo seja salvo com precisão e integridade. Além disso, ocorre o bloqueio na execução de novas requisições. Após o término da operação de *checkpoint*, o processo 1 (p1) retoma sua taxa normal de atendimento de requisições, voltando a estabilizar-se em torno de 250 req/s.

Este gráfico auxilia na compreensão dos impactos que operações de *checkpoint* podem ter em sistemas em tempo real ou sistemas que demandam alta disponibilidade

e vazão. Ele destaca a necessidade de técnicas que minimizem o tempo de inatividade durante *checkpoints* ou que distribuam a carga de trabalho de tal forma que o impacto no desempenho seja reduzido.

Nesta seção serão abordados os principais tipos de *checkpoints* existentes na literatura. Serão apresentados os *checkpoints* coordenados e não coordenados, incrementais e particionados. Também são discutidos os *checkpoints* difusos, os dinâmicos e adaptativos, bem como os *checkpoints* a nível de processo de usuário e a nível de sistema.

2.1.1 Coordenados e não coordenados

Checkpoints coordenados exigem a sincronização entre todas as réplicas participantes do sistema na execução do processo. Neste, as réplicas são sincronizadas ao iniciar a execução do processo de *checkpointing* simultaneamente e aguardam até que todas as réplicas tenham finalizado (JANAKIRAMAN; TAMIR, 1994; CHANDY; RAMAMOORTHY, 1972; TAMIR; SEQUIN, 1984). Essa sincronização faz com que o sistema fique com seu estado bloqueado e não execute operações até que o processo termine, causando um pico na latência do sistema. Essa sincronização serve para garantir a consistência dos *checkpoints* e que todos eles tenham exatamente os mesmos dados.

A Figura 4 apresenta o processo de *checkpoints* coordenados entre dois processos, p1 e p2. Esta representação facilita a compreensão da sequência e coordenação entre os processos durante a criação de *checkpoints* em um sistema distribuído. O eixo horizontal representa o avanço do tempo. À medida que nos movemos da esquerda para a direita, observamos as diferentes etapas do processo de *checkpoint*. As duas linhas horizontais representam a linha do tempo para os processos p1 e p2. O ícone de diamante marca o início da operação de *checkpoint* para ambos os processos. Antes de iniciar o *checkpoint* propriamente dito, os processos passam por fases de *prepare* (preparação) e *startckp* (início do checkpoint). Durante a preparação, os processos se certificam de que estão prontos para iniciar o *checkpoint*. Após estarem prontos, a fase de *startckp* começa. A barra com uma rede quadriculada representa o período durante o qual o processo está efetivamente capturando e salvando seu estado atual. Para p1 e p2, este período é indicado pelo retângulo com padrão quadriculado vermelho. O retângulo laranja quadriculado, para p1, indica um período de espera. Durante este tempo, p1 já concluiu seu *checkpoint* e está aguardando que p2 complete o seu. Após a conclusão do *checkpoint*, cada processo cria um arquivo *foto*, representado pelos ícones de cubo. Este arquivo contém o estado salvo do processo naquele instante. Uma vez que ambos os processos tenham concluído seus *checkpoints* e os arquivos tenham sido criados, eles avançam para a fase de finalização.

Nos *checkpoints* não coordenados, as réplicas não precisam realizar a sincronização (MOSTEFAOUI; RAYNAL, 1996; MENDIZABAL; JALILI MARANDI *et al.*, 2014).

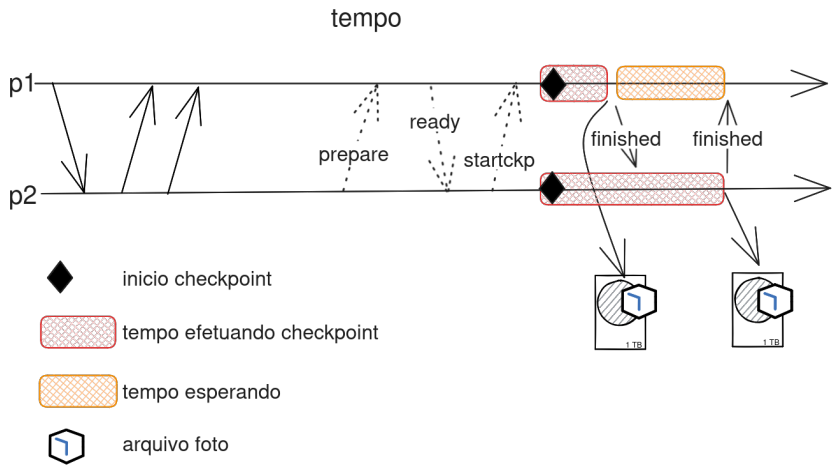


Figura 4 – *Checkpoint* coordenado

Alguns trabalhos buscaram melhorar e propor estratégias para esse tipo de *checkpointing*. A vantagem dele em relação ao anterior é que, enquanto uma das réplicas está realizando o *checkpointing*, as outras réplicas podem continuar atendendo e executando as operações.

Entretanto, o processo de recuperação é mais complexo visto que as réplicas possuem *checkpoints* em momentos diferentes e com estados diferentes. Esta abordagem pode ser útil em situações onde a coordenação entre processos é difícil ou indesejada, mas pode levar a situações em que a recuperação global do sistema após uma falha é complexa Diante disso, o processo de recuperação precisa ter a noção de qual das réplicas tem o *checkpoint* mais recente e disponível para utilização. A Figura 5 ilustra uma situação onde este tipo de *checkpoint* foi utilizado. Cada processo executa o *checkpoint* de maneira independente e não aguarda pelo outro.

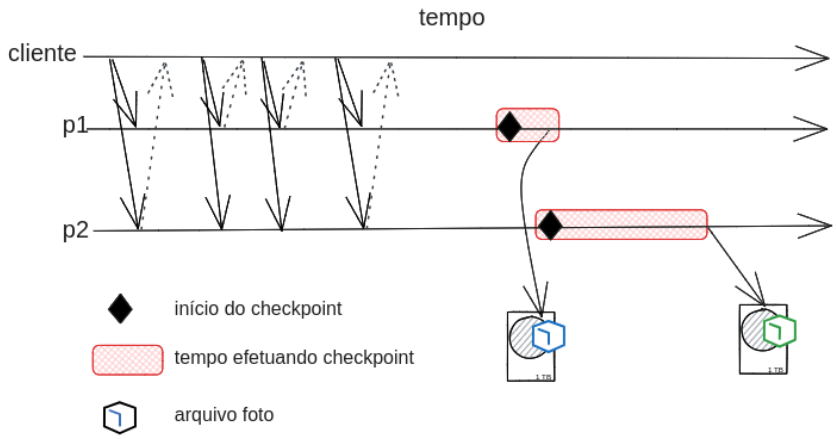


Figura 5 – *Checkpoint* não coordenado

2.1.2 Incremental

Checkpoints incrementais possibilitam a preservação de apenas uma parte do estado do sistema, em contraposição à captura do estado inteiro durante a criação de um *checkpoint*. Na Figura 6 é possível ver que cada *foto* do estado representa apenas uma parte do que estava na memória do processo *p1* depois de sofrer alterações por requisições feitas pelo *cliente*, estando destacado em vermelho quadriculado o que foi salvo em cada um deles.

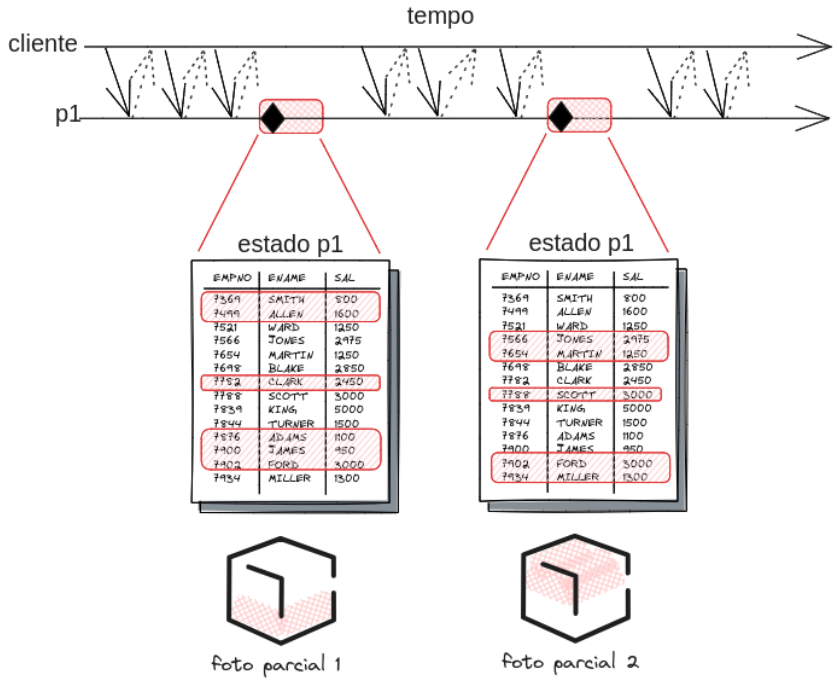


Figura 6 – *Checkpoint* incremental

Tal técnica foca nos elementos do estado do sistema modificados desde o último *checkpoint*, diminuindo a carga associada ao *checkpointing* e o tempo necessário para sua execução (ELNOZAHY; JOHNSON; ZWAENEPOEL, 1992). Apesar de os *checkpoints* incrementais apresentarem vantagens, o armazenamento de múltiplos *checkpoints* pode aumentar o consumo de espaço. Um mecanismo de consolidação ou união de *checkpoints* anteriores evita um crescimento descontrolado. A Figura. 7 ilustra esse problema onde é observado um aumento no número de arquivos contento parcialmente a *foto* do estado.

Além disso, vale mencionar que os *checkpoints* incrementais não são apropriados para todos os tipos de sistemas; uma aplicação que modifica e acessa diversas partes do sistema com frequência torna o processo de *checkpointing* tão custoso quanto o método tradicional de *checkpointing*, que captura o estado total do sistema, além de adicionar a complexidade do *checkpoint* incremental. Um exemplo notório de *checkpointing* incremental é o Libckp (PLANK *et al.*, 1994), que armazena de forma eficiente

somente as páginas modificadas do sistema Linux a partir do último *checkpoint*, aproveitando as capacidades do hardware para identificar e isolar corretamente as partes inalteradas do estado do sistema.

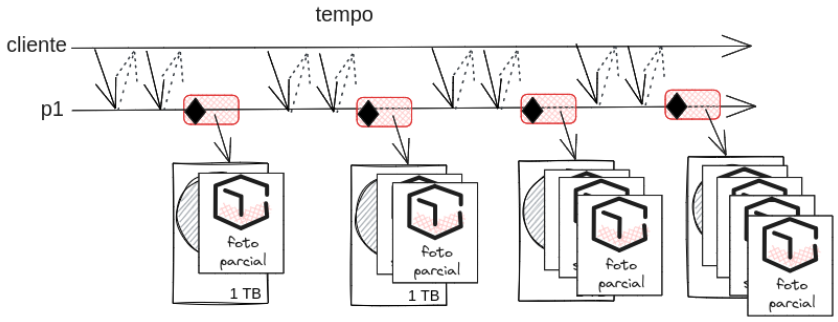


Figura 7 – *Checkpoint* incremental - consolidação

Com o objetivo de paralelizar a execução do *checkpointing* incremental, foram sugeridas algumas técnicas que tiram proveito da função *fork()* do sistema operacional. Essa função cria um processo permitindo que processo pai prossiga com a execução enquanto o processo filho efetua o *checkpointing*. Além disso, essas técnicas implementam a estratégia de cópia na escrita (*copy-on-write*), copiando páginas da memória do processo pai para o filho apenas quando estas sofrerem atualizações, enquanto os dados não modificados continuam sendo compartilhados entre os processos (ELNOZAHY; JOHNSON; ZWAENEPOEL, 1992; LI; NAUGHTON; PLANK, 1994). É importante destacar que a estratégia de cópia na escrita pode ser intensiva em termos de memória, sobretudo em computação científica, em que a memória física do computador é geralmente utilizada ao máximo, restando pouco espaço para alocação adicional de memória.

2.1.3 Particionado e Difuso

A intenção desta abordagem é utilizar o paralelismo existente no hardware atual, tanto em termos de processamento quanto de E/S. A meta é tornar a operação de *checkpointing* paralela, o que pode acelerar o processo de salvar e restaurar o estado do *checkpoint*. *Threads* ou tarefas em paralelo podem lidar com partições de estado distintas do sistema em questão. Contudo, esse método adiciona uma complexidade extra na manutenção da consistência, uma vez que os estados de *checkpoint* são fragmentados em partições menores e não retratam o estado completo do sistema. Normalmente, o processo de recuperação se torna mais difícil quando lida com *checkpoints* divididos. A Figura 8 ilustra duas *threads*, onde cada uma lê e salva apenas uma partição dos dados do estado *Estado p1*. Assim, cada *thread* gera seu arquivo individualmente, paralelizando a escrita e leitura da memória e do dispositivo de armazenamento.

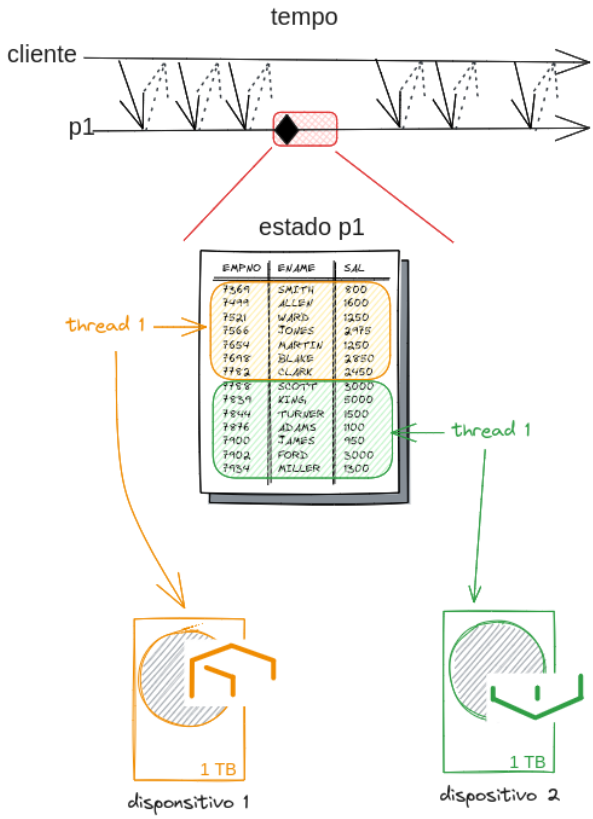


Figura 8 – *Checkpoint* particionado

Em (JUNIOR; ALCHIERI *et al.*, 2023), os pesquisadores sugeriram uma técnica de *checkpointing* que possibilita ao sistema prosseguir com sua execução comum nas partições que não estão envolvidas no *checkpointing*, ao mesmo tempo em que uma partição específica passa pelo procedimento de *checkpointing*. Isso resulta em um *checkpoint* que não ilustra de forma completa o estado do sistema em um instante específico. Tais *checkpoints* com informações parciais são denominados na literatura como *checkpoints* difusos (*fuzzy checkpoints*) e apresentam novos obstáculos para o processo de recuperação.

Um efeito notável na realização de *checkpoints* é a intermitência do sistema, exibindo períodos de inatividade ou, no mínimo, redução da eficiência e aumento da latência. Isso acontece porque os *checkpoints* geralmente precisam armazenar uma versão completa e consistente do estado do serviço em um momento específico. Uma estratégia para diminuir essa sobrecarga durante o funcionamento normal é permitir a execução do processo de *checkpointing* enquanto as operações rotineiras estão modificando parte do estado do serviço. Essa estratégia se beneficia do paralelismo, mas produz *checkpoints* difusos. Na Figura 9 esse comportamento é ilustrado pela marcação em vermelho representando os dados que foram alterados durante a execução do *checkpoint*.

Isso implica que o resultado do *checkpoint* não é, necessariamente, uma versão consistente do estado do serviço em um ponto específico da sequência de execução.

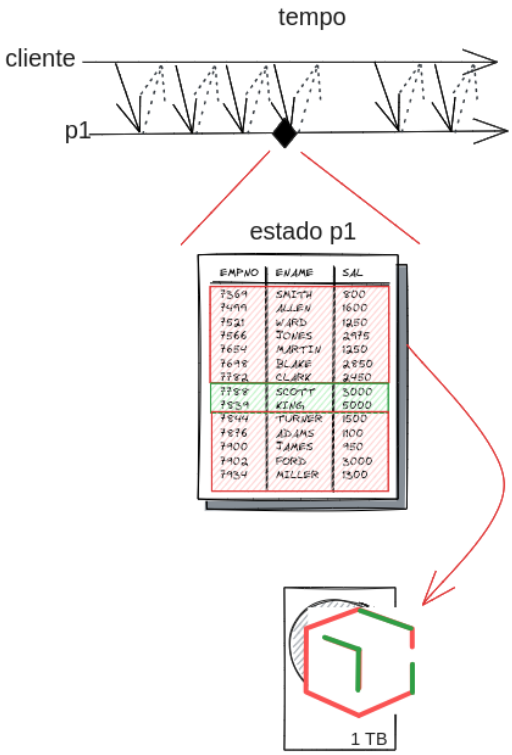


Figura 9 – *Checkpoint* difusos

Sendo assim, para recuperar uma versão consistente, é sempre necessário restaurar um *checkpoint* e reproduzir ao menos uma parte do registro de comandos executados para atualizar informações que não foram salvas pelo *checkpoint*.

Em (ZHENG *et al.*, 2014), os pesquisadores sugeriram um método rápido de durabilidade e recuperação para bancos de dados em memória baseados em *checkpoints* difusos. Os *checkpoints* podem ser realizados em paralelo com a execução normal. Partes do *checkpoint* são armazenadas em diversos arquivos, o que facilita o processo de truncamento de registro, e aproveita a capacidade de vários dispositivos de armazenamento simultaneamente. Uma estratégia semelhante é apresentada em (JUNIOR; ALCHIERI *et al.*, 2023), em que réplicas de serviço adotam um esquema de replicação ativa, e, em intervalos determinados, cada réplica salva o estado de apenas um grupo das partições de estado. Quando a carga de trabalho é adequada, as réplicas podem salvar diferentes partições, permitindo o paralelismo entre os *checkpoints* das réplicas.

2.1.4 Dinâmicos e adaptáveis

Os métodos convencionais de *checkpointing* costumam empregar intervalos fixos para estabelecer o momento adequado para efetuar um *checkpoint*. Contudo, algumas pesquisas indicam que a variação dos períodos de intervalo pode diminuir o tempo total de processamento da aplicação sem comprometer a confiabilidade. Considerar modelos probabilísticos e o conhecimento específico das aplicações ao decidir os momentos apropriados para iniciar um procedimento de *checkpoint* pode melhorar

ainda mais o processo e aprimorar a eficiência do sistema.

Por exemplo, observa-se que é mais provável que uma falha ocorra logo após outra falha (TIWARI; GUPTA; VAZHKUDAI, 2014). Da mesma forma, é possível considerar a probabilidade de falha em determinados contextos, já que um processo pode ter apenas uma probabilidade moderada de falha (FRANK *et al.*, 2021). Ajustando os intervalos de *checkpoint* com base na probabilidade de falha, é possível reduzir o tempo total de computação, evitando *checkpointing* desnecessário e minimizando a computação desperdiçada, que se refere à computação perdida entre o último *checkpoint* e o momento da falha.

A Figura 10 ilustra esse comportamento. O eixo horizontal representa a passagem do tempo, com eventos ocorrendo sequencialmente de esquerda para direita. As duas linhas paralelas superiores representam entidades, um cliente e um processo ou servidor denominado *p1*. Os eventos descritos na linha de tempo refletem as ações ou estados dessas entidades ao longo do tempo. Os losangos ao longo das linhas representam pontos onde são tirados *checkpoints*. Em um sistema real, esses *checkpoints* seriam momentos em que o estado atual do sistema é salvo para recuperação futura, caso ocorra uma falha. Abaixo do eixo do tempo, há várias marcações como $\triangle t$, $\triangle t/2$, $2\triangle t$ e $3\triangle t$. Estes representam intervalos de tempo que podem ser ajustados dinamicamente. Por exemplo, o sistema pode inicialmente salvar *checkpoints* a cada intervalo de $\triangle$t, mas, com base em alguma lógica ou condição, pode ajustar esse intervalo para $\triangle t/2$ ou $2\triangle t$, tornando o processo de *checkpointing* mais ou menos frequente conforme a necessidade.

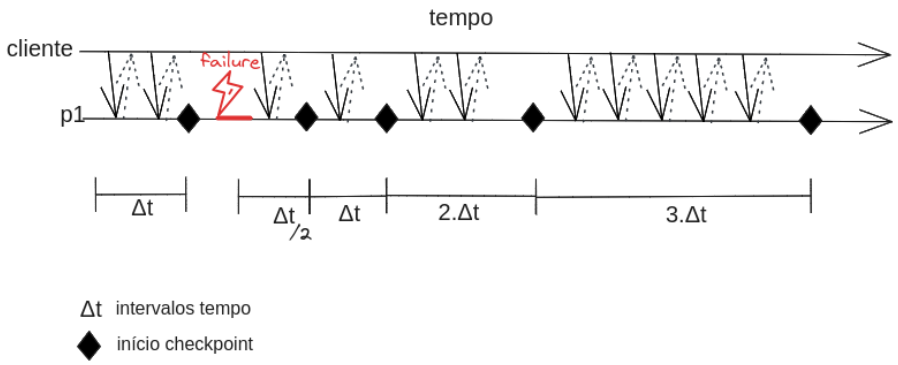


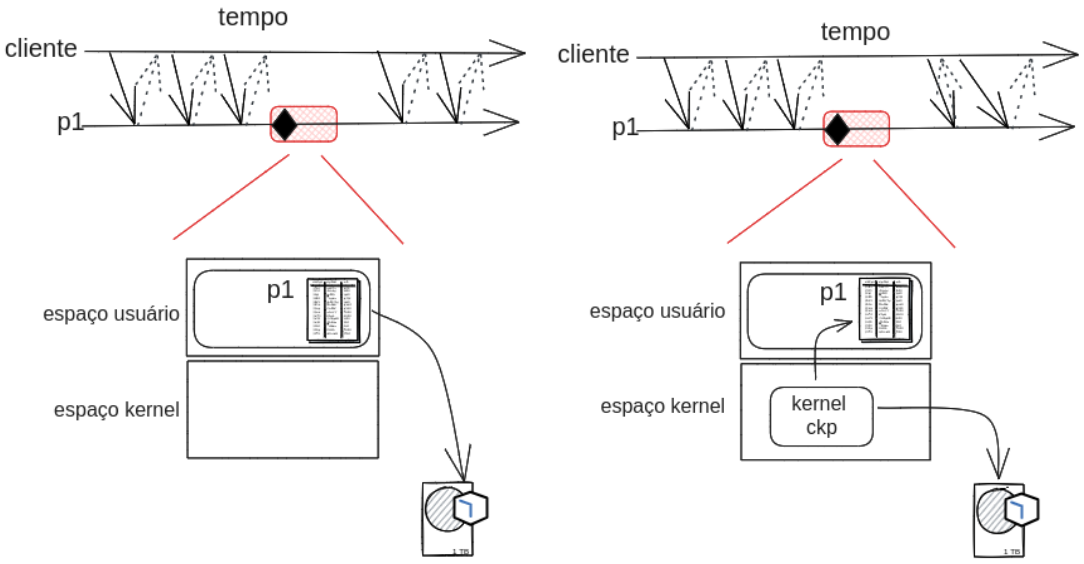
Figura 10 – *Checkpoint* dinâmico

2.1.5 Nível de usuário e nível de sistema

A implementação de *checkpoints* pode ocorrer tanto no nível do núcleo do sistema operacional quanto no espaço do usuário. *Checkpoints* de nível de sistema, ilustrado pela Figura 11b, como o Linux Checkpoint/Restart do Berkeley Lab (BLCR) (DUELL, 2005) e o Linux Checkpoint/Restart como um módulo de *kernel* (CRAK) (ZHONG; NIEH, 2001), fornecem acesso de baixo nível, permitindo a interação direta com os dados

de um processo sem dificuldades. Essa abordagem facilita o rastreamento dos estados do processo e a criação de *checkpoints* consistentes. Além disso, os *checkpoints* de nível de kernel oferecem maior desempenho devido à sua operação direta no sistema operacional. No entanto, como estão integrados ao kernel, podem exigir manutenção à medida que o kernel evolui, limitando potencialmente a portabilidade do código.

No *checkpointing* do espaço do usuário (PLANK *et al.*, 1994; CRIU, 2021), ilustrado pela Figura 11a, o processo de *checkpoint* deve monitorar os sinais do sistema operacional para identificar e rastrear alterações nas regiões de memória, garantindo um manuseio correto para alcançar *checkpoints* consistentes. Diferentemente do *checkpointing* de nível de kernel, o *checkpointing* do espaço do usuário depende da escuta de interfaces de núcleo de sistema e suas chamadas, o que pode torná-lo mais suscetível a falhas e erros, e potencialmente mais lento do que a abordagem de nível de sistema. No entanto, o *checkpointing* do espaço do usuário não requer modificações no núcleo do sistema, o que aumenta sua portabilidade entre diferentes sistemas.



(a) *Checkpoint* nível de usuário (b) *Checkpoint* nível de kernel

Figura 11 – *Checkpoint* em sistema operacional

3 REVISÃO DA LITERATURA

Ao longo desta seção, serão apresentado alguns trabalhos que influenciaram nossa abordagem e visão. Estes trabalhos são reconhecidos por suas contribuições e por abordarem diferentes aspectos do particionamento de dados e da gestão de cargas de trabalho. Assim, estes estudos oferecem considerações importantes sobre o particionamento de estados e *checkpoints*.

3.1 APLICAÇÕES COM ESTADO PARTICIONADO

Diversos estudos e técnicas que exploram a concorrência na execução de comandos para potencializar a vazão do sistema são discutidos em (MARANDI; BEZERRA; PEDONE, 2014; MENDIZABAL; DE MOURA *et al.*, 2017; ALCHIERI; DOTTI; MARANDI *et al.*, 2018; JUNIOR; AVILA, 2020; KOTLA; DAHLIN, 2004). Muitos desses trabalhos fundamentam-se na análise da semântica do comando em execução para efetuar a execução concorrente dos comandos.

A abordagem *Parallel State-Machine Replication (P-SMR)*, conforme proposto em (MARANDI; BEZERRA; PEDONE, 2014) e (MENDIZABAL; JALILI MARANDI *et al.*, 2014), sugere um conjunto de *threads* executoras onde cada *thread* é designada para administrar uma partição específica, assumindo o controle integral do processamento do *checkpoint* e da recuperação da referida partição.

Os comandos são categorizados em dois tipos: paralelo e sincronizante. Quando identificado como paralelo, o comando é processado pela *thread* associada à respectiva partição, e o resultado é comunicado ao cliente. Em contraste, os comandos sincronizantes acionam uma notificação pela *thread* que os recebe, alertando outras *threads* sobre uma modificação em um estado compartilhado. Assim, múltiplas partições são afetadas. As *threads* correspondentes às partições impactadas devem permanecer em espera até a conclusão do processamento principal. Após a resposta ser enviada ao cliente, as demais *threads* são liberadas. A Figura 12 ilustra estes dois tipos de comandos: à esquerda, os comandos paralelos e, à direita, os comandos sincronizantes.

Na abordagem P-SMR, a determinação de dependências entre os comandos, independentemente de estarem baseados em seus argumentos, é uma responsabilidade do serviço implementado nesta arquitetura. Isso permite ao serviço decidir a relação de dependência entre os comandos, enquanto a arquitetura gerencia sua execução.

Um dos principais objetivos dessa abordagem é melhorar a vazão em aplicações que se beneficiam do paralelismo. Isso é relevante para aplicações onde a maioria dos comandos são leituras, como os *name servers*, ou para aquelas que operam com estados particionados, onde os comandos atuam majoritariamente em uma partição específica sem interferir nas demais, a exemplo dos sistemas de arquivos.

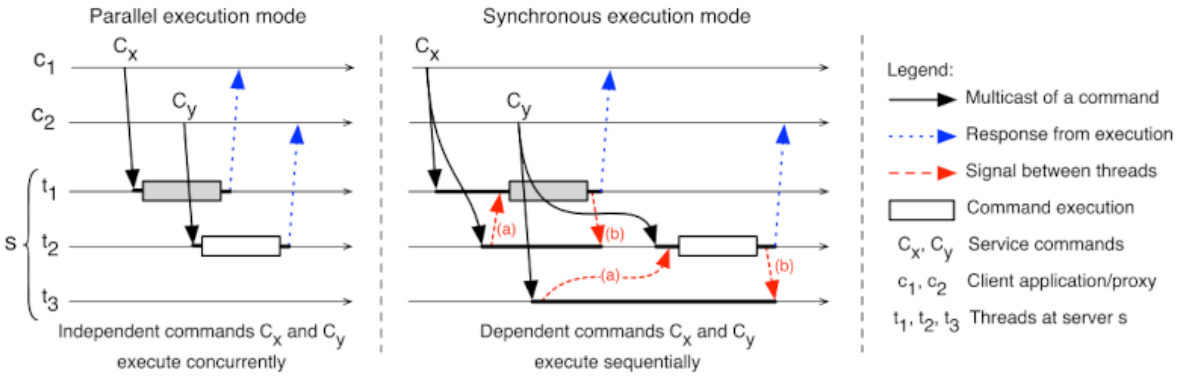


Figura 12 – Dependência de comandos - Adaptado de (MARANDI; BEZERRA; PEDONE, 2014)

3.1.1 Variação de paralelismo

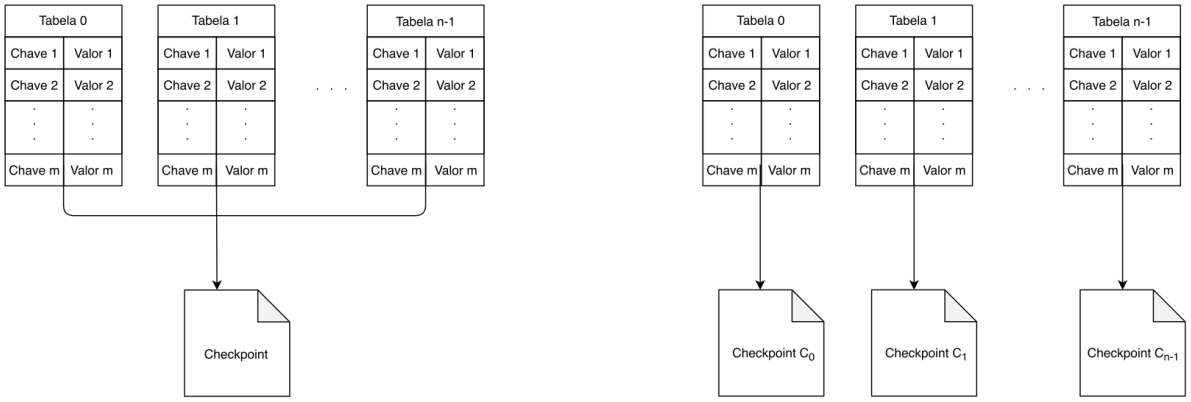
O desempenho do sistema está ligado à quantidade de *threads* em execução. A relação entre o caso de uso e o número de *threads* é significativa. Quando se tem um número reduzido de *threads*, os comandos sincronizantes têm um impacto mínimo na degradação de desempenho, devido às escassas sinalizações de sincronização. Contudo, essa configuração não maximiza a vazão de comandos paralelos. Por outro lado, com um alto número de *threads*, embora o desempenho dos comandos sincronizantes seja afetado negativamente, os comandos paralelos experimentam um aumento considerável na vazão, conforme observado em (ALCHIERI; DOTTI; MENDIZABAL *et al.*, 2017).

O estudo citado em (ALCHIERI; DOTTI; MENDIZABAL *et al.*, 2017) também introduz a ideia de classes de conflito associados a comandos para gerenciar a concorrência. Um comando de uma classe particular pode entrar em conflito com outros da mesma classe ou de classes distintas. O mapeamento das classes de conflitos atribui cada classe de conflito a uma *thread* específica, definindo potenciais conflitos entre classes. Portanto, se um comando de uma *thread* entrar em conflito, a máquina de estado replicado deverá executá-lo de forma sincronizada com as *threads* relevantes. Dessa forma, o trabalho mencionado faz uso da informação semântica dos comandos e, baseado nela, define a alocação. Tal solução demanda que o cliente e o serviço alinhem essa noção semântica de ante-mão para ter essa definição estabelecida estaticamente.

Os trabalhos citados serviram como base para o estudo relacionado ao estado particionado, aplicações e os custos de sincronização. Diante das evidências e estratégias propostas nele, surge, neste trabalho, a opção de efetuar um reparticionamento durante *checkpoint* como uma possibilidade de evitar a dependência de comandos com semântica para se obter uma maior vazão em sistemas particionados.

3.2 CHECKPOINT PARTICIONADO

No estudo apresentado em (JUNIOR; ALCHIERI *et al.*, 2023), uma abordagem de *checkpointing particionado* é introduzida. A meta dessa estratégia é diminuir o tamanho do estado salvo de cada partição, otimizando o tempo de processamento do *checkpoint*. Essa eficiência melhora não apenas o armazenamento dos arquivos de *checkpoint*, mas também a velocidade de leitura dos mesmos. Essa otimização favorece a recuperação rápida de uma réplica defeituosa ou a integração de uma nova réplica ao sistema. O estudo também sugere que os *checkpoints* de diferentes partições sejam realizados em momentos distintos. Isso permite que a réplica continue atendendo às requisições dos clientes nas partições que não estão passando pelo processo de *checkpointing*. A distribuição dos estados em memória de cada partição neste modelo é representada na Figura 13. Vale mencionar que este trabalho é fundamentado no modelo P-SMR, no qual conceitos como *classes de comandos* e *mapeamento de conflitos* são essenciais.

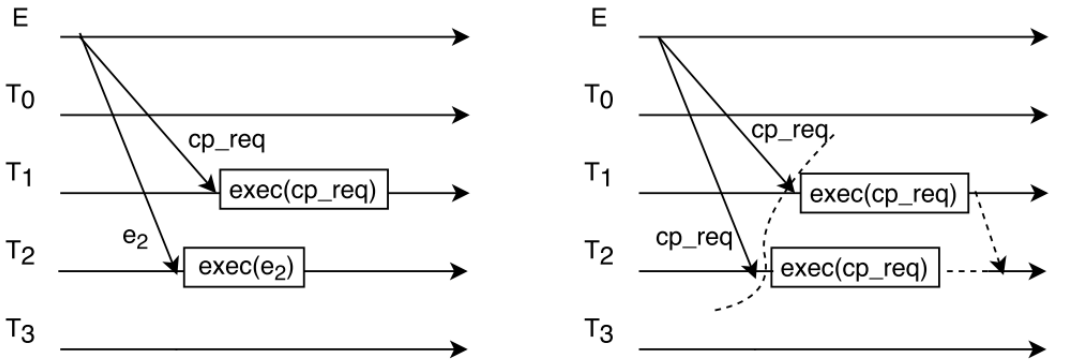


(a) *checkpoint* tradicional (b) *checkpoint* particionado

Figura 13 – Estados particionados particionado - Adaptado de (JUNIOR; AVILA, 2020)

Ainda, é proposto um escalonador responsável pela organização da execução dos comandos. Esse escalonador identifica a *classe de conflito de comando* recebido verificando qual tipo de execução pode ser feita, *paralela* ou *sincronizante*, e então encaminha corretamente para as *threads* responsáveis. Para casos de comandos *sincronizantes*, além da lógica já existente em P-SMR (aquela em que há uma sinalização de aguarde para as *threads* envolvidas), o projeto também armazena uma *matriz de conflito* onde fica registrado quando um comando de uma partição também afetou uma outra partição. Então, dado uma quantidade configurável de comandos executados, o escalonador dispara um comando do tipo *checkpointing* notificando que uma das *threads* deve realizar o processo de salvamento do estado daquela partição. A escolha da *thread* é feita de modo *round-robin*. Antes de o escalonador enviar o comando para a *thread*, este verifica na *matriz de conflito* se alguma outra partição foi afetada por algum comando executado por aquela *thread* em sua partição. Caso isso ocorra,

é disparado um *checkpointing* do tipo *sincronizante* para todas as *threads* afetadas, fazendo com que mais de uma partição efetue o *checkpointing*. Caso contrário, apenas a partição do momento selecionada fará o *checkpointing* e as demais continuam executando comandos. Na Figura 14, o exemplo à esquerda ilustra uma situação onde não é necessária a sincronização das *threads*, visto que elas não fazem parte da *matriz de conflito*. Já no exemplo à direita, observa-se uma situação em que as *threads* t_1 e t_2 estão incluídas na *matriz de conflito*



(a) *Checkpoint* de uma única partição

(b) Sincronização de *threads* para realização de *checkpoints* em paralelo

Figura 14 – Checkpoint particionado - Adaptado de (JUNIOR; AVILA, 2020)

O estudo de (JUNIOR; ALCHIERI *et al.*, 2023) evidenciou um aumento na vazão média do sistema durante fases de *checkpointing*. Também destacou uma diminuição no tempo para integrar uma nova réplica ou recuperar uma réplica comprometida. Entretanto, o estudo pontua que não avaliou o impacto de partições desbalanceadas, e reconhece que isso poderia comprometer a eficácia do modelo. Essa questão fica como sugestão para pesquisas futuras.

Em outra pesquisa, abordada em (BESSANI *et al.*, 2013), o foco é a diminuição do custo associado ao *checkpoint*. Foram conduzidos experimentos em um sistema prático de replicação de máquina de estado, que integra mecanismos como armazenamento de estado em memória persistente, *logging*, *checkpointing* e transferência de estado entre réplicas. O objetivo era identificar os principais gargalos na replicação de máquina de estado. Foram avaliadas métricas como tempo de escrita do *log* no disco, duração do processo de *checkpointing*, seu reflexo no tempo de resposta às solicitações do cliente e, finalmente, os custos de processamento e de rede ao recuperar o estado de uma réplica ausente, utilizando o estado de uma réplica ativa.

Um dos desafios identificados foi o elevado custo de processamento para o *logging*, especialmente ligado ao dispositivo de armazenamento persistente. A latência significativa nas operações de E/S desse dispositivo resulta em uma vazão menor de operações por segundo e um tempo de resposta aumentado.

Diferentemente dos dois trabalhos mencionados acima, este efetua o *checkpoint* de maneira coordenada produzindo assim um estado completo do sistema num dado

momento ao invés de pedaços de *fotos* de momentos diferentes, facilitando assim a posterior recuperação. Além disso, este trabalho foi capaz de produzir partições balanceadas endereçando um dos problemas para trabalhos futuros mencionados em (JUNIOR; ALCHIERI *et al.*, 2023).

3.3 SCHISM

O estudo de (CURINO *et al.*, 2010) apresenta o Schism, uma abordagem de particionamento de banco de dados que leva em consideração a carga de trabalho. Esse documento destaca os custos elevados envolvendo transações que se estendem por múltiplas partições. É apontado que, em sistemas distribuídos atuais, as técnicas mais frequentemente usadas para particionamento são *round-robin*, *range* e *hash-partitioning*.

Em situações onde instruções de banco de dados englobam várias partições localizadas em nós distintos, com cada nó gerindo uma partição, observa-se um declínio no desempenho comparado a um sistema unificado. A principal causa desse declínio é o emprego de protocolos específicos, como o de efetivação em duas fases ou de consenso distribuído, que asseguram a integralidade das operações. O uso desses protocolos torna o processo mais intricado, reduzindo a agilidade do sistema.

A proposta do Schism é baseada em uma técnica que usa grafos para particionar, buscando minimizar os custos e equilibrar a carga. As entradas do banco de dados são simbolizadas como vértices nesse grafo, e as relações entre elas indicam o acesso conjunto a várias entradas durante as transações. Cada acesso a uma entrada resulta no aumento do valor do ponto correspondente no grafo. O estudo propõe duas abordagens: uma que considera o tamanho da partição e outra que se baseia na frequência de acessos.

Uma das maiores preocupações do trabalho é a gestão de grande volume de dados. Isso porque o grafo se amplia à medida que o banco de dados e a complexidade das transações crescem. Embora os algoritmos de particionamento baseados em grafos sejam escaláveis, sua eficácia decresce com o crescimento do grafo. Para resolver essa questão, técnicas de amostragem foram implementadas, otimizando o tamanho do grafo sem prejudicar sua qualidade. Especialmente, as amostragens focadas nas transações e nas entradas mostraram-se particularmente eficientes.

Para a construção do grafo, os pesquisadores desenvolveram um *script* que analisa as transações realizadas no banco de dados e, a partir delas, elabora o grafo. Eles não especificam o momento ideal para realizar essa operação, indicando que pode ser realizado tanto online, durante as transações, quanto offline.

Também, o Schism é reconhecido por suas contribuições no âmbito do particionamento de bancos de dados, mas possui suas limitações. Entre os desafios enfrentados está a expansão das tabelas de roteamento com o crescimento do mapeamento de

tuplas para partições, o que torna a consulta a essas tabelas mais custosa. Ainda, os investimentos iniciais para o particionamento, juntamente com os custos de manutenção, podem se tornar significativos. Entretanto, métodos de particionamento baseados em hash aleatório destacam-se pelo equilíbrio de carga que proporcionam.

O Schism tem a capacidade de criar *scripts* SQL para transferir dados entre partições. Apesar de sua versão atual focar no particionamento de um único banco de dados, há planos de adaptá-lo para reparticionamento. O que chama a atenção é a necessidade de migrar os dados entre partições usando o *script* SQL, o que acarreta um custo operacional considerável. Em contrapartida, nosso estudo propõe utilizar os intervalos de *checkpoint* para realizar o reparticionamento, minimizando o impacto operacional de manutenção.

3.4 SWORD

O artigo (QUAMAR; KUMAR; DESHPANDE, 2013) introduz o SWORD, um *framework* elaborado para enfrentar desafios no particionamento de bancos de dados. Ele adota uma estratégia bifásica para otimizar a administração de registros. Inicialmente, há uma compressão do *hipergrafo* por meio de particionamento *hash* ou função equivalente. Posteriormente, esse *hipergrafo* comprimido é particionado, aprimorando as tabelas de mapeamento, essenciais para encaminhar transações de maneira adequada.

O SWORD é flexível às flutuações de carga, detectando e reagindo a mudanças significativas incrementalmente. Assim, escolhe estrategicamente os conjuntos de dados para migração, focando em períodos de baixa demanda e buscando reduzir transações distribuídas.

Seu mecanismo de reparticionamento incremental mitiga decréscimos de desempenho provenientes de oscilações de carga, sem a necessidade de um reparticionamento completo. Adicionalmente, o SWORD integra um mecanismo de replicação que é sensível à carga de trabalho, solucionando conflitos e otimizando a distribuição de dados. O sistema também conta com um mecanismo de roteamento eficiente, que limita o número de partições consultadas e aperfeiçoa a gestão de memória com tabelas de roteamento mais compactas.

Contudo, apesar das inovações do reparticionamento incremental abordado no artigo (QUAMAR; KUMAR; DESHPANDE, 2013), ponderamos que os intervalos de inatividade durante as operações de E/S no *checkpointing* poderiam ser empregados para realizar um reparticionamento integral. Tal abordagem simplificaria a implementação, descartando etapas como o reparticionamento incremental e outros recursos específicos do SWORD.

3.5 DYPART

O framework DYPART (LI; XU; KAPITZA, 2018) foi desenvolvido para o particionamento dinâmico de estados em protocolos de Tolerância a Falhas Bizantinas (BFT) com execução paralela de solicitações (KOTLA; DAHLIN, 2004; KAPRITSOS *et al.*, 2012). Seu objetivo principal é ajustar em tempo real as partições de estados de aplicativos para otimizar o desempenho, agrupando e dividindo estados.

Durante a operação, cada réplica monitora ativamente as interdependências das solicitações, formando um gráfico que ilustra as conexões entre os objetos de estado. O DYPART coordena operações distribuídas por múltiplas partições, chamadas de solicitações transfronteiriças, assegurando uma execução precisa. A renovação do particionamento é sincronizada ao mecanismo de *checkpoint*, garantindo consistência e reduzindo excessos operacionais.

Quando se alcança o ponto designado para o *checkpoint*, um novo é estabelecido, e todas as réplicas aplicam o algoritmo de particionamento gráfico. Isso visa conservar a frequência de solicitações transfronteiriças ao mínimo, otimizando a eficácia do processamento paralelo.

Em um paralelo com o DYPART, nossa proposta tira proveito dos momentos onde o processador não é utilizado em sua capacidade máxima, durante os *checkpoints*, para realizar o reparticionamento do estado. No entanto, a pesquisa carece de uma análise abrangente e mostra diferenças quando comparada ao nosso trabalho. O DYPART não apresenta uma análise detalhada dos impactos da combinação de *checkpointing* e reparticionamento. O estudo usou apenas uma carga de trabalho para avaliação experimental e discussão, focando principalmente em representar relações entre objetos acessados que retratam a interação entre personagens em *Les Misérables*, fornecido pela Stanford Graph Base (KNUTH, 1993). Em contraste, nosso estudo formaliza o problema de otimização e realiza avaliações experimentais usando cargas de trabalho realistas para avaliar os efeitos da combinação de *checkpointing* e reparticionamento em um contexto mais amplo.

Pesquisas relacionadas a técnicas de *checkpoint* possuem uma vasta literatura, abrangendo sistemas operacionais, bancos de dados, computação de alto desempenho, computação em nuvem e sistemas distribuídos e confiáveis. Embora adotemos uma estratégia de *checkpoint* paralelo particionado, outras abordagens, como *checkpoints* difusos ou incrementais, poderiam ser aplicadas sem limitar a generalidade da abordagem. Portanto, consideramos melhorias na estratégia de *checkpoint* como ortogonais a este trabalho. Em (ELNOZAHY; ALVISI *et al.*, 2002), os autores apresentam o contexto e as principais estratégias para checkpointing em sistemas de troca de mensagens. O checkpointing em HPC é abordado em (EGWUTUOHA *et al.*, 2013) e, em (GOULART; FRANCO; MENDIZABAL, 2023), os autores discutem a evolução das técnicas de *checkpoint* ao longo das últimas décadas.

4 *CHECKPOINT* PARALELO COM REPARTICIONAMENTO

Neste trabalho, propomos uma estratégia de reparticionamento para aplicações com estado particionado durante a execução de um *checkpoint* particionado coordenado. O objetivo central é aproveitar o potencial computacional frequentemente subutilizado devido à intensa operação de E/S associada ao processo de *checkpoint*. Ao fazer isso, podemos implementar um algoritmo de reparticionamento eficiente.

Nas seções subsequentes, são detalhadas as técnicas e os conhecimentos inerentes à solução sugerida. É explorada a dinâmica do estado particionado e os desafios a ele associados. Posteriormente, examinaremos a aplicabilidade dos grafos como ferramentas para gerenciar dados em aplicações de estado particionado. Por fim, será descrito um modelo do problema de otimização que buscamos resolver, esclarecendo os aspectos do problema que visamos explorar.

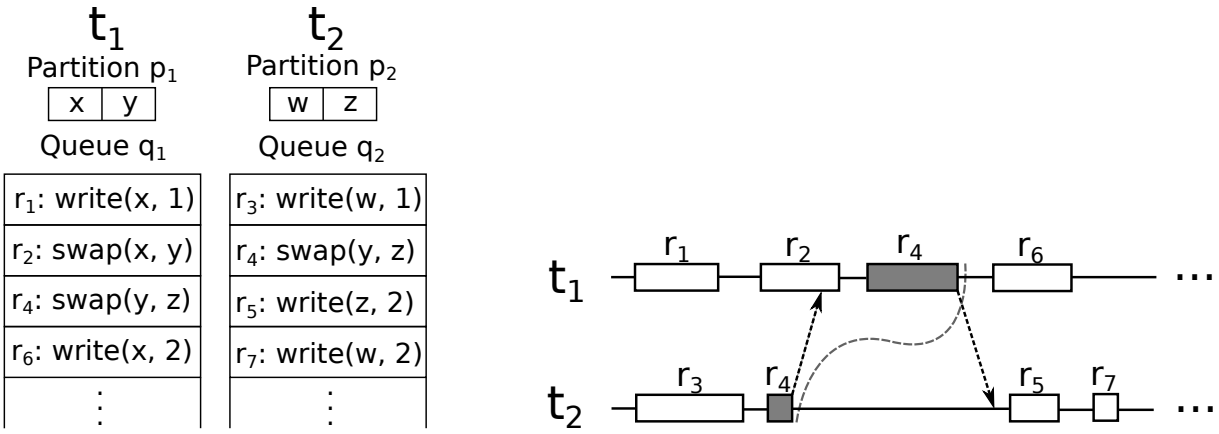
4.1 ESTADO PARTICIONADO

Nesta seção, é explorado um conceito chamado execução particionada. Imagine que se tem um estado geral de uma aplicação, simbolizado por S. Este estado é então segmentado em vários conjuntos distintos, que chamamos de *partições*. Cada uma dessas partições é denotada como p_i, com i variando de 1 até n (número de partições). Portanto, se você combinasse todas as partições, obteria o estado S original. O sistema usa uma *thread* de trabalho para cada partição, ou seja, para a partição p_i, temos uma *thread* específica t_i para gerenciá-la.

Requisições, denotadas por r_i, definem as interações com as chaves. A relação entre as chaves e as requisições é dada por $K(r)$, e cada chave pertence a uma partição específica, representada por $P(k)$. Assim, cada *thread* tem sua própria fila, representada por q_i, para lidar com as requisições destinadas à sua partição. Uma requisição é enviada para a fila de uma *thread* quando se relaciona com uma chave que pertence à partição gerenciada por essa *thread*. As requisições são categorizadas de duas formas distintas: requisições que lidam com uma única variável e aquelas que englobam múltiplas variáveis. No caso das requisições de uma única variável, estas interagem com apenas um elemento do estado da aplicação de maneira atômica. Por outro lado, requisições que acessam múltiplas variáveis estendem-se a duas ou mais variáveis. Este último grupo pode ser ainda mais diferenciado, baseando-se no número de partições envolvidas, seja uma partição singular ou várias partições (TROMBETA *et al.*, 2021).

Nas Figuras 15a e 15b, o estado S é composto por quatro variáveis: x, y, w e z. E têm-se duas *threads*, t_1 e t_2. A primeira *thread* gerencia as variáveis x e y, enquanto a segunda lida com w e z. A certa altura, várias requisições são feitas, e elas são alocadas nas filas das *threads* com base nas variáveis com as quais interagem.

Ao examinar as requisições, observa-se que algumas, especificamente r_1, r_3, r_5, r_6, r_7, tratam somente de uma variável, não exigindo coordenação entre as *threads*. Em contraste, a requisição r_4 se destaca por sua complexidade, interagindo com múltiplas variáveis de diferentes partições e necessitando de sincronização para garantir a integridade do sistema. Como resultado, r_4 é alocada nas filas de ambas as *threads*, t_1 e t_2. No decorrer do processamento, quando t_2 começa a trabalhar em r_4, ela é colocada em estado de espera até que t_1, *thread* com um identificador mais baixo, esteja pronta para processar a requisição. Este mecanismo assegura que r_4 seja processada apenas por t_1, após o qual ambas as *threads* prosseguem com suas tarefas regulares. É importante destacar que, durante a espera de t_2 pela execução de r_4 por t_1, ela permanece inativa, ilustrando a necessidade de coordenação efetiva nas operações que envolvem várias partições.



(a) Fila de comandos e threads

(b) Execução de commandos

Figura 15 – Execução em estado particionado - adaptado de (TROMBETA *et al.*, 2021)

4.2 CHECKPOINTS PARTICIONADOS

O principal objetivo desta abordagem é melhorar o uso do paralelismo em computadores modernos, seja em processamento ou em E/S. Ao paralelizar a ação de *checkpointing*, espera-se agilizar tanto o salvamento quanto a recuperação do estado armazenado. Para isso, utilizamos várias *threads* na estratégia de *checkpointing*. Cada uma delas é designada a salvar uma fração específica do estado global. O agendador de execução, por sua vez, desencadeia o *checkpoint*, inserindo periodicamente um *pedido de checkpoint* em todas as filas das partições. Esse pedido é *multi-variável* envolvendo múltiplas partições, promovendo a sincronização de todas as *threads* para um *checkpointing* conjunto e simultâneo. A operação só é considerada completa quando todas as *threads* finalizam seus respectivos processos.

Revendo a Figura 8 (Checkpoint particionado) apresentada na Seção 2.1.3, observa-se que ambas as *threads* podem ser processadas simultaneamente, melhoran-

do o tempo de *checkpoint*. Em sistemas com múltiplos dispositivos de armazenamento, o ganho é ainda maior com o uso de E/S de vários dispositivos ao mesmo tempo, aumentando a eficiência ao criar uma *foto* do estado do sistema. O tempo total de um *checkpoint* é, por fim, determinado pela *thread* que demora mais a concluir sua tarefa. Cada *checkpoint*, agora, é formado por múltiplos arquivos, com cada um refletindo a *foto* de uma partição distinta. No âmbito da recuperação, este esquema particionado é igualmente vantajoso, permitindo a restauração paralela de arquivos. Apesar de não aprofundarmos a temática da recuperação neste trabalho – dado seu caráter intuitivo –, vale destacar que a restauração completa do estado ocorre pela reintegração de todos os arquivos de *checkpoint*.

4.3 GRAFOS

A eficácia do particionamento de estado, que promete paralelismo, depende diretamente de como o agendador aloca as requisições nas *threads*. Uma alocação ineficaz pode resultar em sincronizações custosas (ALCHIERI; DOTTI; MENDIZABAL *et al.*, 2017). Criar um particionamento de estado que atenda às necessidades do serviço é complexo, dada a necessidade de antever a demanda de trabalho. Além disso, mesmo munido de informações relevantes, as tendências de acesso aos dados podem variar. Isso significa que uma configuração local eficiente pode enfrentar desafios com flutuações na demanda geral. Portanto, reajustes dinâmicos, realizados durante a execução, surgem como solução para manter o balanceamento de carga entre as partições.

Quanto às estratégias de particionamento, os algoritmos de corte balanceado em grafos destacam-se. Eles são utilizados em diversas áreas, desde o mapeamento de rotas, redes computacionais, até a estruturação de bases de dados. Neste contexto, um grafo pode representar o acesso em um estado segmentado em múltiplas partições.

A Figura 16 ilustra a representação de um estado particionado em grafo, destacando duas ou mais partições (*partição1*, ... *partiçãoN*). Essas partições são delineadas por retângulos coloridos, azul para *partição1* e verde para *partiçãoN*.

Dentro de cada partição, observam-se nós, que representam estados individuais baseados na chave de acesso de uma requisição, e arestas, que simbolizam cômputo de operações *multi-variável*. Os nós estão identificados por letras (x, y, k, c, w, z e h). Os números em vermelho identificam os pesos e a quantidade de acessos efetuados em cada nós e aresta.

Na partição *partition1*, o nó x tem uma aresta conectando-o ao nó k, com peso 2, indicando que houve 2 acessos a essas chaves em operações simultâneas, como *swap/scan*. O nó k, por sua vez, tem um peso 7, indicando que houve 7 acessos à chave que ele representa. Essa mesma lógica se aplica aos demais nós e arestas do grafo. Entre as duas partições, há uma aresta com peso 4, representando um acesso

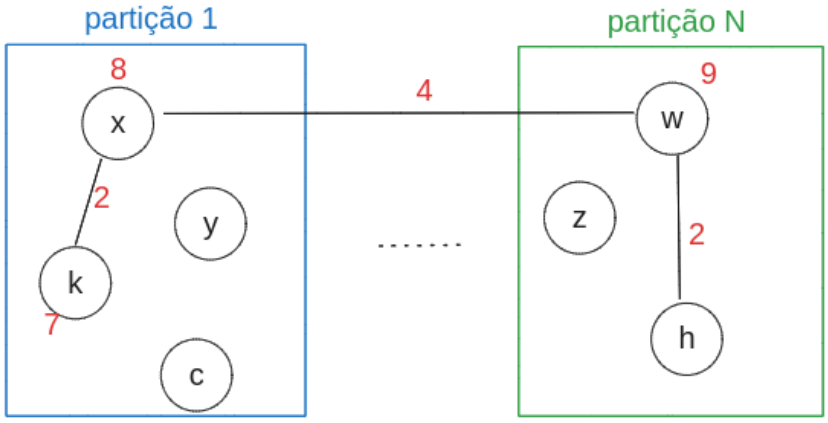


Figura 16 – Estado representado com um grafo

entre-partições. Esta aresta simboliza a interação ou comunicação entre estados de diferentes partições, o que requer uma coordenação de acesso entre eles.

4.4 PROBLEMA DE OTIMIZAÇÃO

A capacidade de executar tarefas de forma paralela, graças à divisão de estado, é promissora. No entanto, o sucesso da execução das tarefas depende muito de como o agendador organiza e distribui essas demandas para as *threads*. Quando essa organização não é eficiente, o sistema pode enfrentar sincronizações que consomem muito tempo e recursos (ALCHIERI; DOTTI; MENDIZABAL *et al.*, 2017). Uma das grandes dificuldades é determinar a melhor forma de dividir o estado do serviço, especialmente porque precisa-se prever a carga de trabalho que virá. E mesmo quando se tem uma ideia das demandas futuras, os padrões de acesso aos dados não são fixos e podem evoluir. Dada essa natureza mutável, adaptar e reorganizar a divisão das tarefas em tempo real pode ajudar a distribuir a carga de trabalho de forma mais eficiente.

Neste trabalho, é utilizado de grafos para tratar o problema do balanceamento das partições e reduzir o número de operações *multi-variável entre-partições*, diminuindo assim a necessidade de sincronizações. No balanceamento, busca-se uma distribuição equilibrada dos estados entre as partições para aprimorar a operação de *checkpoint*. Nos parágrafos a seguir será definido o problema e em seguida a sua implementação.

4.4.1 Definição do problema

Tem-se um sistema que monitora como cada variável de estado é usada. Ele observa quantas vezes essas variáveis são solicitadas para leitura ou escrita, ou quando várias delas são solicitadas por uma mesma operação. Toda essa informação é armazenada em um *grafo ponderado*. Os pesos desses vértices, representados por c_i para o vértice i, indicam quantas vezes essa variável foi solicitada para operações de leitura ou escrita.

Agora, quando duas ou mais variáveis são acessadas juntas, temos uma aresta no grafo entre cada par de variáveis correspondente. O peso dessa aresta, representado por w_{ij} para a aresta que liga i e j, nos diz o quão frequentemente essas duas variáveis são usadas em conjunto.

Ao longo deste capítulo, denote por $G = (V, E)$ o grafo obtido através das requisições recebidas pelo sistema, onde V é o conjunto de vértices e E o de arestas. Com este grafo em mãos, o sistema pode começar a agrupar as variáveis em conjuntos ou partições. Cada grupo de variáveis é chamado de p_i em um conjunto de partições $P = \{p_1, \ldots, p_k\}$. O ideal é que as variáveis em uma partição específica, digamos p_i, sejam usadas com frequências semelhantes, e essa frequência, ou peso total, seja comparável ao de outras partições, como p_j. Além disso, deseja-se minimizar o número de requisições que utilizam variáveis de partições diferentes, para evitar sincronizações e ineficiências.

Neste cenário, o principal objetivo é tornar o processo de armazenamento de dados paralelo mais rápido e também reduzir a necessidade de sincronização das requisições. Este desafio, de agrupar as variáveis da maneira mais eficiente possível, é tecnicamente conhecido como o *problema de corte mínimo de partição equilibrada (balanced partition minimum cut problem)*. É um problema NP-completo (GAREY; JOHNSON; STOCKMEYER, 1974) e muitas heurísticas são propostas para resolvê-lo (NISHIMURA; UGANDER, 2013; TSOURAKAKIS *et al.*, 2014; KARYPIS; KUMAR, 1998; SANDERS; SCHULZ, 2013).

A estratégia que é colocada por este trabalho divide o problema em duas fases. Na primeira, o foco é em criar partições eficientes tendo tamanhos similares. Na segunda, minimizar a comunicação entre as partições. Com base nessa estratégia, espera-se obter um sistema equivalente e baseado em fases.

4.4.1.1 Fase 1

Inicia-se com a definição de uma variável inteira, y, que atua como uma métrica de equilíbrio na distribuição dos vértices. O objetivo é identificar um valor ideal para y, que seja o menor possível, sinalizando assim um eficiente balanceamento de dados. Este aspecto é observado no lado direito da inequação (2).

São introduzidas variáveis binárias x_{ip} que indicam a relação entre vértices e partições:

$$x_{ip} = \begin{cases} 1, & \text{quando o vértice } i \text{ é alocado à partição } p, \\ 0, & \text{em outras situações.} \end{cases}$$

Na estruturação proposta, define-se um valor constante k. Assume-se a existência de k partições, as quais são representadas pelo conjunto $P = p_1, \ldots, p_k$. Define-se C como a somatória dos pesos dos vértices, representada por $C = \sum_{i \in V} c_i$. O desafio é distribuir os vértices em P de maneira que nenhuma partição fique sobrecarregada.

$$
\begin{aligned}
&\min\ y \\
&\text{sujeito a:}\quad \textstyle\sum_{p\in P} x_{ip} = 1, \forall i \in V && (1)\\
&\qquad\quad \textstyle\sum_{i\in V} c_i x_{ip} \leq \frac{C}{k} + y, \forall p \in P && (2)\\
&\qquad\quad x_{ip} \in \{0,1\}, \forall i \in V \text{ e } \forall p \in P && (3)\\
&\qquad\quad y \in \mathbb{Z}^+. && (4)
\end{aligned}
$$

A restrição (1) garante que cada vértice seja alocado em uma e somente uma partição. A restrição (2) assegura que nenhuma partição esteja sobrecarregada na solução ótima, uma vez que nosso objetivo é minimizar o valor de y. Já as restrições (3) e (4) definem que as variáveis x_{ip} devem ser binárias e que y deve ser um número inteiro, respectivamente.

4.4.1.2 Fase 2

Com o valor da variável y já determinado na *Fase 1*, definimos $M = \frac{C}{k} + y$, a próxima etapa busca identificar o corte mínimo levando em consideração esse valor específico para y. Enquanto mantemos as variáveis x_{ip}, introduzimos novas variáveis binárias, representadas por z_{ijp}.

$$
z_{ijp} = \begin{cases} 1, & \text{se a aresta } i-j \text{ contém os vértices } i \text{ e } j \text{ na} \\ & \text{partição } p, \\ 0, & \text{caso contrário.} \end{cases}
$$

$$
\begin{aligned}
&\min \quad \textstyle\sum_{i-j\in E} w_{ij}(1 - \sum_{p\in P} z_{ijp}) \\
&\text{sujeito a:}\quad \textstyle\sum_{p\in P} x_{ip} = 1, \forall i \in V && (5)\\
&\qquad\quad \textstyle\sum_{i\in V} c_i x_{ip} \leq M + b, \forall p \in P && (6)\\
&\qquad\quad z_{ijp} \leq x_{ip}, \forall i-j \in E \text{ e } \forall p \in P && (7)\\
&\qquad\quad z_{ijp} \leq x_{jp}, \forall i-j \in E \text{ e } \forall p \in P && (8)\\
&\qquad\quad z_{ijp} \geq \tfrac{1}{2}(x_{ip} + x_{jp}) - \tfrac{1}{2}, \forall i-j \in E, \\
&\qquad\qquad\qquad\qquad \forall p \in P && (9)\\
&\qquad\quad x_{ip} \in \{0,1\}, \forall i \in V \text{ e } \forall p \in P && (10)\\
&\qquad\quad z_{ijp} \in \{0,1\}, \forall i-j \in E. && (11)
\end{aligned}
$$

As restrições (5) e (6) garantem uma partição equilibrada. Já as restrições (7), (8) e (9) identificam as arestas dentro das partições. As restrições (10) e (11) especificam que as variáveis x_{ip} e z_{ijp} são binárias.

Dada a restrição (5), que coloca cada vértice i em exatamente uma partição p, para uma aresta $i-j$, ambos os vértices estão em uma partição p (quando i e j estão em p) ou, no máximo, um deles está em p.

A restrição (6), originária da Fase 1, foi modificada pela adição da variável b. Esta variável introduz uma flexibilidade ao modelo com o objetivo de possibilitar a identificação de configurações de partição que, mesmo não sendo as melhores sob

o critério da Fase 1, possam resultar em cortes mais eficientes quando aplicadas à Fase 2. Assim, b funciona como um parâmetro de ajuste que expande o número de soluções admissíveis, focando não apenas na otimização isolada da Fase 1, mas também na qualidade do corte na Fase 2, promovendo um balanceamento entre as fases do modelo. Se ambos os vértices estiverem em p, z_{ijp} é determinado como 1 pela restrição (9). Caso contrário, z_{ijp} é definido como 0 pelas restrições (7) ou (8). Portanto, para uma aresta fixa $i - j$, a soma $\sum_{p \in P} z_{ijp}$ é igual a 1 apenas quando ambos os vértices i e j estão na mesma partição p. Assim, a soma dos pesos das arestas entre as partições pode ser calculada conforme descrito na função objetivo do problema da Fase 2.

É importante observar que qualquer solução viável do problema da Fase 1 pode ser usada para construir uma solução viável para o problema da Fase 2. Dada uma solução viável y e x_{ip} para o primeiro problema, as variáveis x podem ser mantidas no segundo problema e definimos $z_{ijp} = 1$ para qualquer aresta ij apenas se $x_{ip} = 1$ e $x_{jp} = 1$ (ou $z_{ijp} = 0$, caso contrário).

4.4.2 Solução do problema

Embora o processo de *checkpoint* implique custos elevados devido às intensas operações de E/S, ele permite um processamento adicional durante sua execução. Percebendo essa subutilização do processador, sugerimos uma abordagem que diminui o impacto dos custos de repartição: executá-la paralelamente com o *checkpoint*.

Assim que se inicia o processo de *checkpoint*, o sistema também dá início à repartição. Ambos os processos, uma vez concluídos, permitem que o sistema retorne à sua operação normal. Para garantir uma repartição eficiente, o agendador rastreia as solicitações e suas partições associadas. O agendador também mantém a estrutura de dados que relaciona cada variável com a partição a que pertence, facilitando a análise do comportamento da carga de trabalho.

O agendador mapeia os acessos de partição usando uma estrutura de grafos, na qual cada comando de acesso é associado a um vértice. Cada vez que uma partição é acessada, o valor de seu vértice correspondente é incrementado. Em comandos que envolvem várias variáveis, como $swap(a, b)$, cria-se uma aresta entre os vértices de a e b. Acessos similares subsequentes amplificam o valor dessa aresta. Vale destacar que, neste modelo, uma aresta representa a relação entre duas variáveis. Portanto, um comando que envolva três variáveis será representado por três arestas, cada uma ligando um par de variáveis.

O processo de repartição se baseia nesse grafo mantido pelo agendador, representando assim todo o contexto da carga de trabalho. Utilizando os valores dos vértices e arestas, o algoritmo rearranja o grafo de modo a equilibrar o peso entre todas as partições. O algoritmo também considera o *custo de corte* de cada partição, referente

às arestas que interligam partições distintas. A meta principal é minimizar esse custo de corte, assegurando que os pesos das partições mantenham-se equilibrados.

5 IMPLEMENTAÇÃO E AVALIAÇÃO DE DESEMPENHO

Neste capítulo, é apresentada uma avaliação experimental da abordagem proposta. Foi desenvolvido um protótipo em C++ para armazenar pares de chave-valor, incorporando uma estratégia de *checkpoint* paralelo sincronizante. O protótipo foi adaptado de (TROMBETA; MENDIZABAL, 2020) incluindo controles de *checkpoints* particionados sincronizantes bem como outras lógicas de funcionalidades necessárias para o devido funcionamento e testes.

Na próxima seção, será apresentada a arquitetura do protótipo. Em seguida, descreveremos a metodologia dos experimentos. Por fim, serão expostos e analisados os resultados obtidos com a adoção da técnica de *checkpoint* particionado com rebalanceamento.

5.1 PROTÓTIPO

O protótipo implementa um armazenamento em memória para os pares de *chave-valor*, *threads* de partição com filas de requisições a serem executadas, e um agendador responsável pela distribuição das requisições. Após atingir um determinado número n de requisições, as *threads* de *checkpoint* e o reparticionamento são iniciados simultaneamente.

Na Figura 17, a seta rotulada como *requisições* indica o ponto de entrada do sistema. Estas requisições são distribuídas às filas das *threads* correspondentes, com base em um grafo mantido pelo agendador. Este grafo associa cada *chave* de requisição a uma *thread* específica. No caso de uma *chave* desconhecida, ela é alocada a uma *thread* através de um algoritmo de *round-robin*, e essa associação é então registrada no grafo. Após a execução de n requisições, as *threads* de *checkpoint* são ativadas e o processo de reparticionamento é iniciado. Esta ativação se dá pelo enfileiramento de uma requisição do tipo *startCheckpoint*, gerada pelo agendador. Durante a criação das *fotos* do estado atual pelo *checkpoint*, o agendador inicia o reparticionamento de forma paralela. Ao finalizar o reparticionamento, o grafo original no agendador é substituído por um novo, já rebalanceado. O sistema então aguarda a notificação das *threads* de *checkpoint*, indicando a conclusão do processamento. Recebida essa notificação, o agendador retoma a execução, enfileirando novas requisições nas *threads* de execução

As *threads* de execução são inicializadas junto com o sistema, permanecendo em espera até que recebam requisições em suas respectivas filas. Uma vez que as requisições estão disponíveis, as *threads* identificam o tipo de operação a ser executada, que pode variar entre leitura, escrita, notificação de sincronização, leitura de varredura de múltiplas variáveis. Em casos de requisições simples e de variável única, a *thread* realiza a operação modificando o dado no estado de sua partição

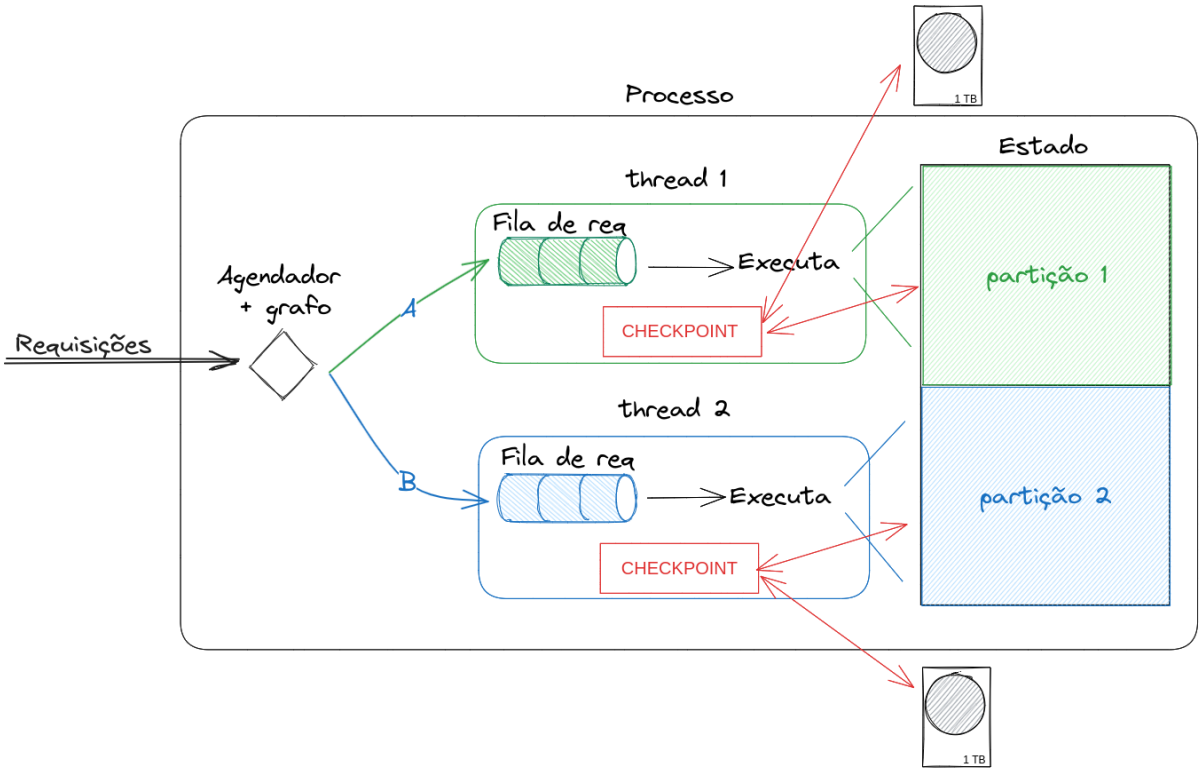


Figura 17 – Arquitetura do protótipo

para escritas, ou lendo o valor da memória associado à chave naquela partição e retornando o dado. No entanto, para operações mais complexas, como leitura de varredura de múltiplas variáveis, é necessária a sincronização prévia entre as *threads*. Esta sincronização é orquestrada por uma requisição de sincronização gerada pelo agendador, que identifica quando uma operação exige tal coordenação, para todas as *threads* envolvidas na operação.

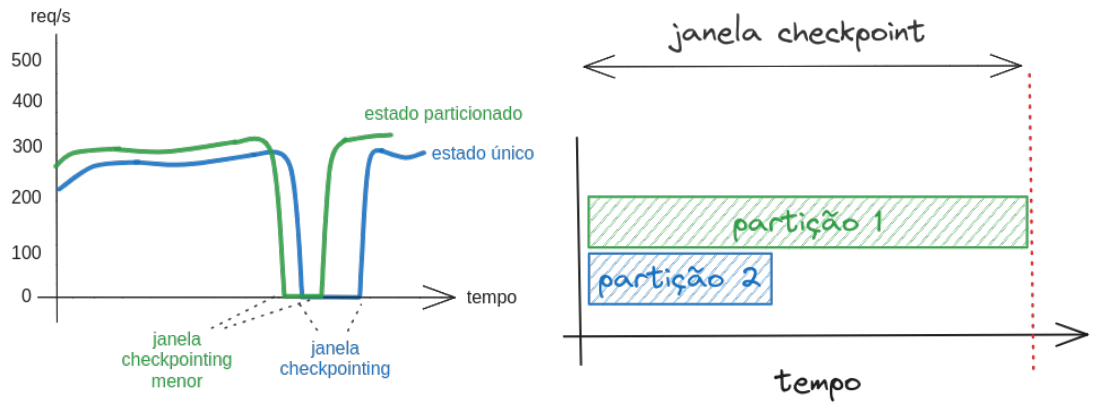
O protótipo foi desenvolvido inicialmente em (TROMBETA *et al.*, 2021) para avaliação do desempenho do particionamento de estado em Replicação Máquina de Estados Paralela. Neste trabalho, ele foi adaptado ganhando funcionalidades de *checkpoint* e sincronizando esse processamento de *checkpoint* com o processamento de reparticionamento. Assim, ele possui todas as funcionalidades para que se possa efetuar os experimentos e avaliações da ideia proposta. Também foram adicionadas métricas de rastreamento em todas as *threads* para entender o efeito de executar o reparticionamento durante o processamento de *checkpoint*. Além disso, também é configurável qual a carga de trabalho que será executada, quantas *threads* executoras e de *checkpoint* devem ser *instanciadas*, e qual o intervalo de requisições executadas para iniciar um *checkpoint*. O protótipo está disponível na Internet[1].

Ainda na Figura 17, são apresentadas apenas duas instâncias de *threads* de *checkpoints* por uma questão de simplicidade no desenho. No entanto, a implementa-

[1] KVPaxos - https://github.com/hensg/kvpaxos - branch: siplified-replica

ção é configurável, permitindo a existência de mais de duas partições, com cada uma possuindo sua própria *thread* de *checkpoint*. As *partições* representam as divisões de dados dentro do sistema. No diagrama, são apresentadas duas partições: *partição 1* e *partição 2*. As *threads* de *checkpoint* interagem diretamente com estas partições, garantindo que os dados nelas contidos sejam consistentemente salvos e atualizados.

Com base na arquitetura do protótipo, espera-se obter uma vazão mais elevada durante a operação normal do sistema, com uma redução nas requisições *multi-variável entre-partições*, conforme ilustrado na Figura 18a. Mesmo considerando o custo computacional para manter o grafo do *scheduler* em memória, pretende-se alcançar uma vazão semelhante para operações que não envolvem tais requisições. Além disso, busca-se que a janela de *checkpoint* seja otimizada, com uma interrupção mais curta no atendimento de requisições. Isso se deve ao balanceamento das partições, prevenindo cenários como o apresentado na Figura 18b, já que a duração da janela de *checkpoint* é dada pela partição mais lenta a realizar o *checkpoint*.



(a) Vazão e janela de checkpoint (b) Tamanho das partições

Figura 18 – Execução em estado particionado com checkpoint particionado

O algoritmo METIS, detalhado na seção subsequente, foi adotado para o reparticionamento realizado pela *repartition thread*. Este foi integrado ao protótipo e configurado para operar em conjunto com o *checkpoint*. Escolheu-se este algoritmo porque ele se alinha à definição (4.4.1) e solução (4.4.2) do problema apresentado neste estudo.

5.1.1 METIS

Dentro deste contexto de reparticionamento e otimização, o problema de corte mínimo em uma partição equilibrada surge como uma necessidade presente em várias áreas da ciência da computação. Para abordar este desafio, recorremos ao METIS, um algoritmo de particionamento de grafos multinível reconhecido por sua eficácia e relevância acadêmica (KARYPIS; KUMAR, 1998). O METIS opera em três etapas.

Na *primeira etapa*, o foco é na simplificação do grafo. O objetivo é reduzir o número de vértices para acelerar o processo subsequente. Através de um algoritmo de correspondência aleatória, os vértices são selecionados para serem colapsados em um único vértice multimodal (BUI; JONES, 1993). Durante essa fase, as arestas e vértices são combinados para formar um grafo simplificado, e as arestas do novo vértice são atribuídas com base nos pesos das arestas originais. Uma técnica chave utilizada aqui é o método *Heavy Edge Matching* (HEM), que se concentra em mesclar vértices associados a arestas de alto peso.

A *segunda fase* é onde ocorre o particionamento do grafo simplificado. Mantendo os pesos dos vértices e arestas previamente combinados, o grafo é particionado inicialmente em duas partes, e em iterações subsequentes, essas partes são divididas ainda mais. Este processo iterativo continua até que sejam alcançadas as k partições desejadas. O particionamento é facilitado pelo uso de um algoritmo de bissecção espectral, que se baseia nos valores da matriz Laplaciana (BARNARD; SIMON, 1994; POTHEN; SIMON; LIOU, 1990).

Finalmente, na *terceira etapa*, chamada de refinamento, o algoritmo recupera o grafo original, mantendo as informações de particionamento. Esse processo envolve uma reconstrução meticulosa do grafo, desde o mais simplificado até sua estrutura original. Durante esse estágio, técnicas de busca local, como o método Kernighan-Lin (KL) (KERNIGHAN; LIN, 1970), são empregadas para garantir que as partições estejam de tamanhos similares.

5.2 EXPERIMENTOS

A análise foca no processamento de requisições, omitindo as camadas cliente e de rede para simplificação. As requisições, pré-carregadas na memória, são distribuídas pelo *scheduler*. O experimento utilizou um computador com Intel i5-11400F, 32GB de RAM e 1TB de armazenamento NVMe da Samsung (980 PRO m2).

Este estudo avalia a influência do *checkpoint* e da divisão de tarefas no desempenho do sistema. Ao tratar uma solicitação, os dados são segmentados em oito partes, adotando-se a técnica round-robin para alocação de chaves. Mantêm-se oito *threads* para operações regulares, ajustando-se o número de *threads* de *checkpoint* conforme a necessidade. A escolha por oito *threads* baseou-se nas especificações do processador, que comporta doze *threads* em seis núcleos, considerando também as funções do *scheduler* e outras exigências do sistema.

Foram investigadas quatro estratégias de execução diferentes:

1. Distribuição round-robin sem *checkpoint* e repartição;
2. Distribuição round-robin com oito *threads* executoras mas apenas uma de *checkpoint*, ou seja, sem paralelismo de *checkpoint*;

3. Distribuição round-robin com oito *threads* executoras e de *checkpoint*;
4. Distribuição inicial round-robin com oito *threads* executoras e de *checkpoint*, e aplicação de reparticionamento.

Há um enfoque particular na estratégia que implementa o algoritmo de repartição, buscando melhorar a utilização dos recursos de processamento durante o *checkpoint*.

5.2.1 Carga de trabalho

Utilizamos o Yahoo! Cloud Serving Benchmark (YCSB) (COOPER *et al.*, 2010) com o objetivo de gerar variadas cargas de trabalho que simulam situações reais. Especificamente, focamos em avaliar o desempenho de diferentes configurações do protótipo sob três perfis de carga:

- YCSB-A: é composto por requisições de variável única, com 50% de leituras e 50% de escritas.
- YCSB-D: 5% das requisições são de inserções e os restantes 95% de leituras, com ênfase nas chaves que foram recentemente inseridas.
- YCSB-E: 95% das requisições são leitura de varreduras de múltiplas chaves, representando requisições multi-variáveis. Os 5% restantes são inserções. Nota-se que, nesse perfil, as varreduras abordam até 8 chaves simultaneamente. Este destaca-se dos demais por envolver operações que acessam múltiplas chaves em uma única requisição, e estas chaves podem estar tanto na mesma partição quanto em partições distintas.

Para melhor clareza sobre os perfis de carga, apresentam-se alguns exemplos. O perfil de carga YCSB-A pode ser representado por um sistema de sessão que armazena as ações recentes do usuário em um site. No caso da carga YCSB-D, um exemplo seria a atualização de mensagens de usuário em redes sociais, enfatizando a leitura das mensagens mais recentes. Finalmente, a carga YCSB-E se assemelha a um fio/*thread* de conversas em redes sociais, onde cada *scan* corresponde a uma mensagem inicial e suas interações subsequentes.

Optamos por excluir as cargas de trabalho YCSB-B e YCSB-C de nossa análise, pois previámos que seus resultados seriam semelhantes ao perfil YCSB-A.

Quanto aos detalhes das configurações para as cargas de trabalho:

- YCSB-A envolveu 94 milhões de requisições com 8 milhões de chaves distintas.
- YCSB-D contou com 110 milhões de requisições e 10 milhões de chaves distintas.
- YCSB-E teve 6 milhões de requisições relacionadas a 2 milhões de chaves distintas.

As cargas de trabalho foram calibradas para garantir aproximadamente dois minutos de execução para cada uma e para assegurar que a memória do computador fosse suficiente para processá-las. Essa calibração resultou em diferenças no número de requisições executadas em cada caso.

Finalmente, para avaliar o desempenho, focamos em métricas como vazão (*throughput*), *makespan*, duração dos processos de criação de *checkpoint* e reparticionamento, tamanho dos *checkpoints* e a distribuição geral das solicitações.

5.2.2 Vazão de execução

Na Figura 19 é ilustrado a vazão ao longo do tempo, representando o número de requisições processadas a cada segundo. O estudo se baseia na carga de trabalho YCSB-A, notória por seu volume intenso de atualizações. Esta configuração divide igualmente suas operações: 50% para escrita e 50% para leitura, sendo que todas as operações acessam uma única variável.

O gráfico apresenta quatro configurações:

- *no-ckp*: usada como referência, sem realização de *checkpoint*.
- *1p-ckp*: executa um único de *checkpoint* não particionado por uma *thread*.
- *8p-ckp*: realiza *checkpoint* em paralelo utilizando oito threads.
- *8p-ckp-metis*: além de executar de *checkpoint* paralelamente com oito *threads*, também incorpora o algoritmo METIS para reparticionamento.

Observando essa carga, nota-se que a ativação do primeiro *checkpoint* ocorre imediatamente antes dos 20 segundos, evidenciado pela redução na vazão. A queda é mais marcante na configuração *1p-ckp*, que retém uma vazão de zero por cerca de 50 segundos, em contraste com as versões paralelas. As versões *8p-ckp* e *8p-ckp-metis* apresentam períodos mais curtos de inatividade (aproximadamente 20 segundos), ressaltando a eficiência de sua estratégia de *checkpoint*.

Na comparação entre *8-ckp* e *8-ckp-metis*, percebe-se que a integração do algoritmo de repartição traz um custo adicional em operações regulares, levando a uma ligeira redução na vazão (cerca de 700k requisições/s). Contudo, o tempo para finalizar o *checkpoint* é similar em ambas, como ilustrado na Figura 32. Uma inspeção detalhada no código revela que essa redução é devido a código adicional relacionado à manipulação de grafo, necessário para monitorar os acessos para reparticionamento.

A Figura 20 mostra as mesmas métricas da visualização anterior, porém para a carga de trabalho YCSB-D. Essa carga é definida pelo seu padrão de acessar o dado mais recente, o que significa que os dados mais recém-inseridos são os mais frequentemente acessados.

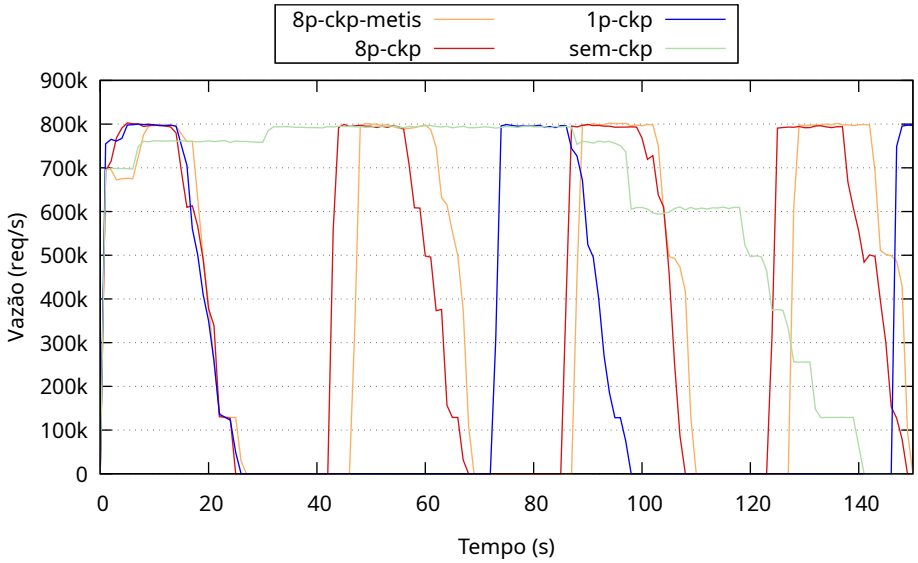


Figura 19 – Vazão de execução - YCSB-A

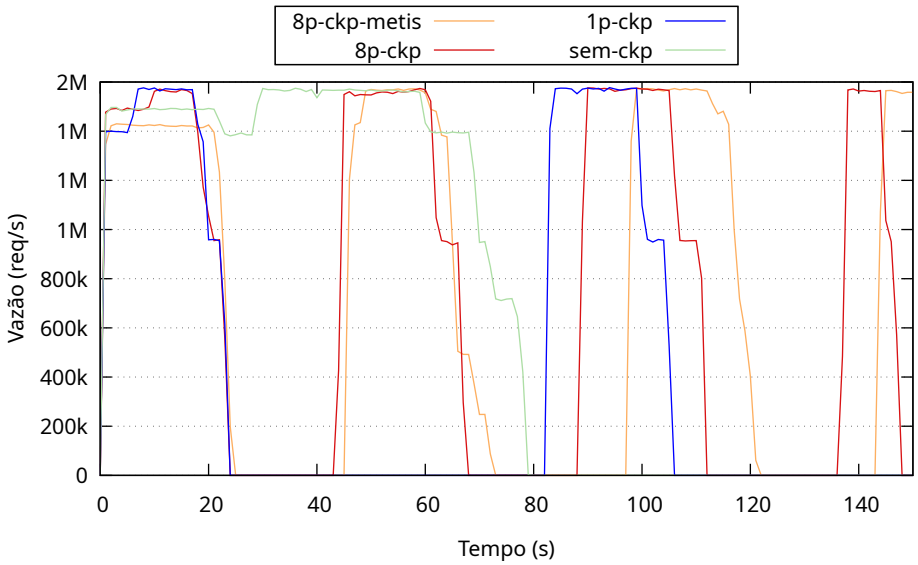


Figura 20 – Vazão de execução - YCSB-D

Os resultados mais surpreendentes dos experimentos estão relacionados à carga de trabalho YCSB-E. Esta se diferencia de outras cargas de trabalho por suas operações acessarem diversas variáveis de dados simultaneamente. Especificamente, a YCSB-E é marcada por operações que varrem faixas de chaves de 1 a 8, indicando tanto operações que acessam apenas uma chave quanto aquelas que abrangem até oito chaves. Predominantemente, essa carga de trabalho foca em operações de leitura, representando 95% das ações, enquanto as escritas compõem os 5% restantes.

A vazão da YCSB-E é notavelmente menor comparada a outras cargas, conforme ilustrado na Figura 21. Esse decréscimo no desempenho deve-se ao aumento do custo de leitura de múltiplas variáveis em diferentes partições e à necessária sincronização entre *threads* relacionadas.

A mencionada figura destaca um crescimento na vazão após a aplicação de nossa técnica de *checkpoint*. Esse aumento é influenciado pelo algoritmo de reparticionamento, responsável por agrupar variáveis acessadas concomitantemente na mesma partição. O experimento revela que, mesmo os *checkpoints* particionados levando um tempo semelhante para serem processados, a abordagem com reparticionamento (*8p-ckp-metis*) apresenta avanços significativos se comparada com aquelas sem essa técnica (*8p-ckp*), alcançando um incremento de cerca de 2,5 vezes na vazão.

Vale destacar que, mesmo as configurações *no-ckp*, *1p-ckp* e *8p-ckp* operando com oito *threads* para partições, seu desempenho é similar. Tal limitação é causada pelas operações de varredura de faixa, que necessitam da sincronização de várias threads, uma vez que as variáveis em questão estão localizadas em partições distintas.

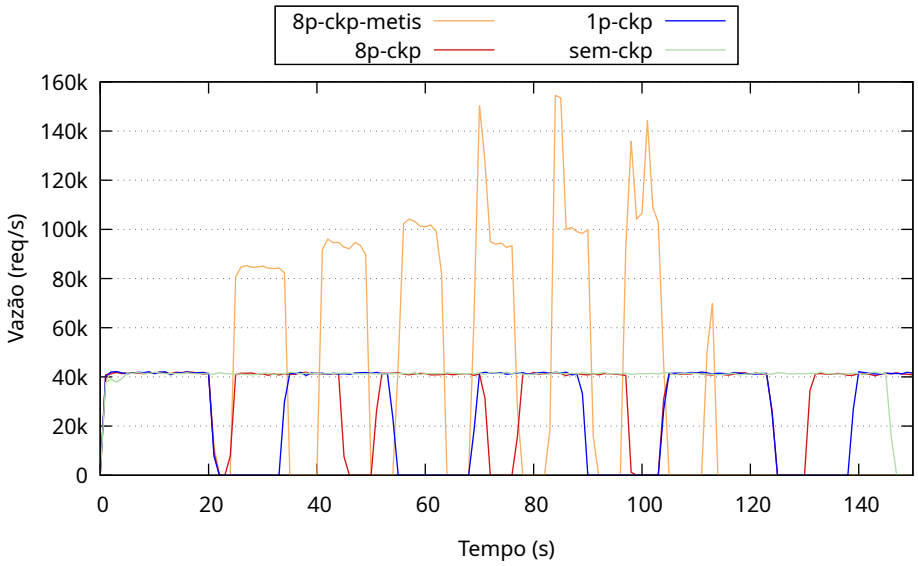


Figura 21 – Vazão de execução YCSB-E.

5.2.3 Acessos multi-variável entre-partições

No monitoramento realizado observou-se a quantidade de *threads* de trabalho em ação: variava de uma única (quando todas as variáveis estavam agrupadas na mesma partição) até oito (quando as variáveis estavam distribuídas em oito partições separadas). A Figura 22 ilustra como essa distribuição de acesso ocorre na carga de trabalho YCSB-E durante as operações de varredura.

A configuração *8p-ckp-metis* chama atenção por seu comportamento distinto. Nela, a maior parte das operações de varredura acessa chaves que estão localizadas na mesma partição. Esse fator contribui para um aumento na vazão, já que há uma redução na necessidade de sincronização entre diferentes partições. Importante notar que na figura está representado o número de partições acessadas e não o identificador da partição como em outras figuras.

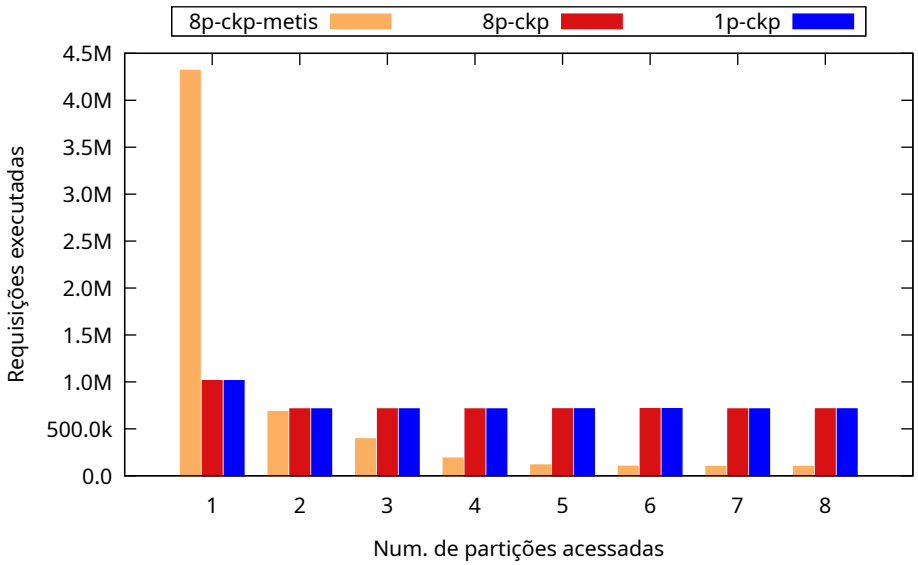


Figura 22 – Cross-border acesso YCSB-E.

5.2.4 Distribuição de carga

No conjunto de resultados a seguir, o termo *distribuição* refere-se à quantidade de solicitações executadas por cada partição, sendo o eixo x associado à *thread*. É importante notar que a configuração *8p-ckp-metis* se destaca por alcançar uma distribuição mais equilibrada para as cargas de trabalho YCSB-A e YCSB-E, conforme ilustrado nas Figura 23 e Figura 25. Em contrapartida, essa configuração não conseguiu equilibrar eficientemente a carga de trabalho YCSB-D, conforme evidenciado na Figura 24, muito em virtude da singularidade das chaves nas requisições recentemente inseridas.

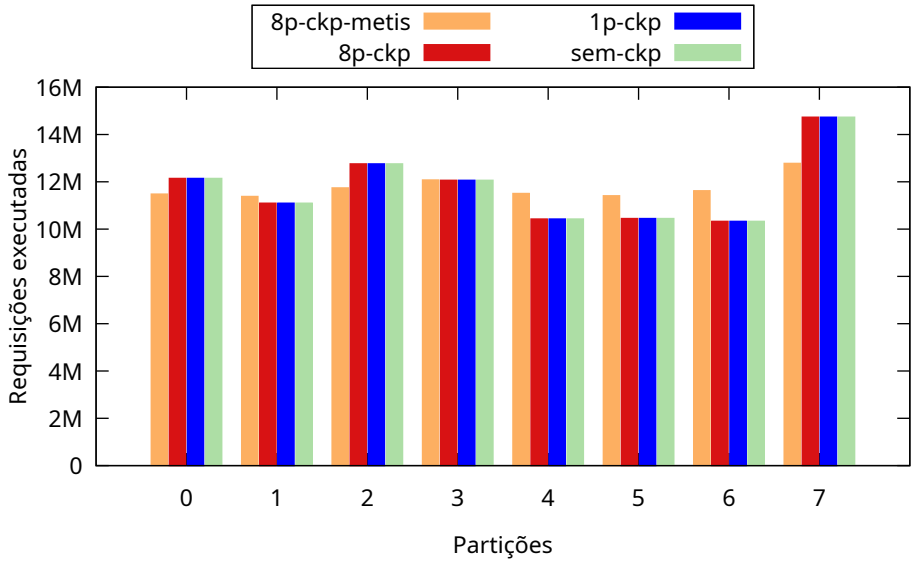


Figura 23 – Distribuição de acesso YCSB-A

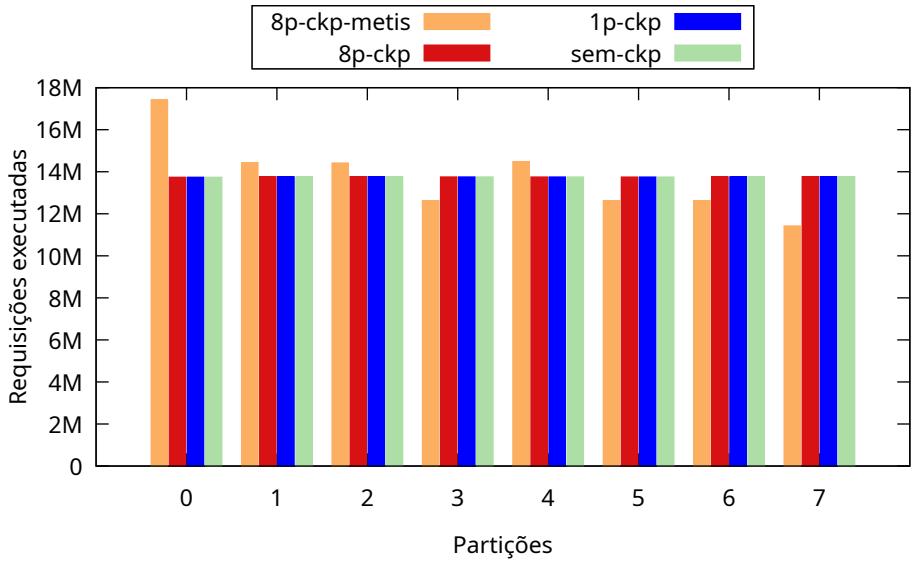


Figura 24 – Distribuição de acesso YCSB-D

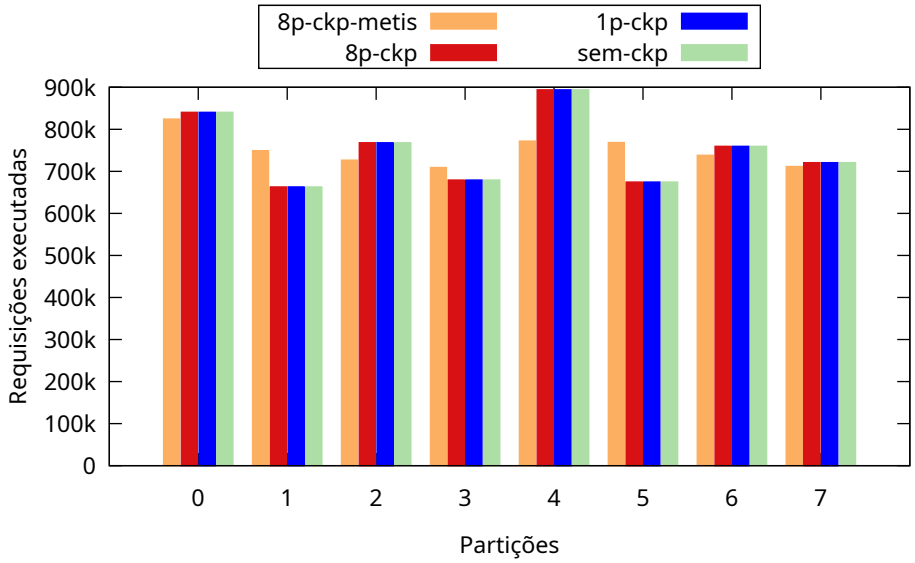


Figura 25 – Distribuição de acesso YCSB-E

Quando ocorre um desequilíbrio na distribuição das tarefas, algumas *threads* de partição podem demorar mais para concluir suas execuções, resultando em outras *threads* em estado ocioso. Esse atraso, especialmente nas *threads* que lidam com maior volume de dados, impacta negativamente o tempo total dos *checkpoints* paralelos. Apesar dos esforços do algoritmo de reparticionamento para equilibrar a distribuição, desafios persistem na criação de um modelo de particionamento otimizado, em particular com a carga de trabalho YCSB-D, que tem variações frequentes nas chaves mais acessadas.

Neste contexto, uma estratégia adaptativa de reparticionamento poderia ser benéfica. Permitiria ao sistema avaliar continuamente sua eficiência e realizar ajustes somente quando houvesse benefícios claros em desempenho. Assim, poder-se-ia me-

lhorar a eficiência e reduzir o trabalho extra e desnecessário, aumentando a robustez do sistema para cargas de trabalho em constante mudança.

5.2.5 Tamanho de *checkpoint* e duração

Os tamanhos dos *checkpoints* correspondem ao tamanho de cada partição de *checkpoints*, refletindo o tamanho médio, em GB, de todos os *checkpoints* executados por essa partição ao longo dos testes. As Figuras 26, 27 e 28 oferecem uma comparação entre duas configurações em relação aos tamanhos das partições. A configuração *8p-ckp* adota a abordagem *round-robin* para distribuir as chaves entre as partições, representando o melhor esquema de partição para distribuição de estado e, portanto, serve como referência. Já a estratégia *8p-ckp-metis* é dinâmica e altera variáveis entre partições conforme a carga de trabalho.

As Figuras 26 e 28 destacam as distribuições de tamanho de partição mais divergentes sob a configuração *8p-ckp-metis*. Essa disparidade nos tamanhos das partições é uma consequência não intencional da nossa abordagem de particionamento. Nossa estratégia busca equilibrar o acesso a variáveis entre as partições e minimizar interrupções de sincronização de solicitações multi-variáveis, ao mesmo tempo em que tenta manter um equilíbrio aproximado entre as partições.

As linhas horizontais representam as partições maiores para *8p-ckp-metis* (cor laranja) e *8p-ckp* (cor vermelha). Conforme observado, um particionamento distribuído de maneira ideal pelo esquema round-robin resulta em partições de estado de cerca de 4 GB para a carga de trabalho YCSB-A. Porém, com a nossa abordagem, uma partição corresponde a 5GB. Ao avaliar a carga de trabalho YCSB-E, notamos uma partição de 1,5 GB, em comparação com 1,1 GB das partições perfeitamente equilibradas.

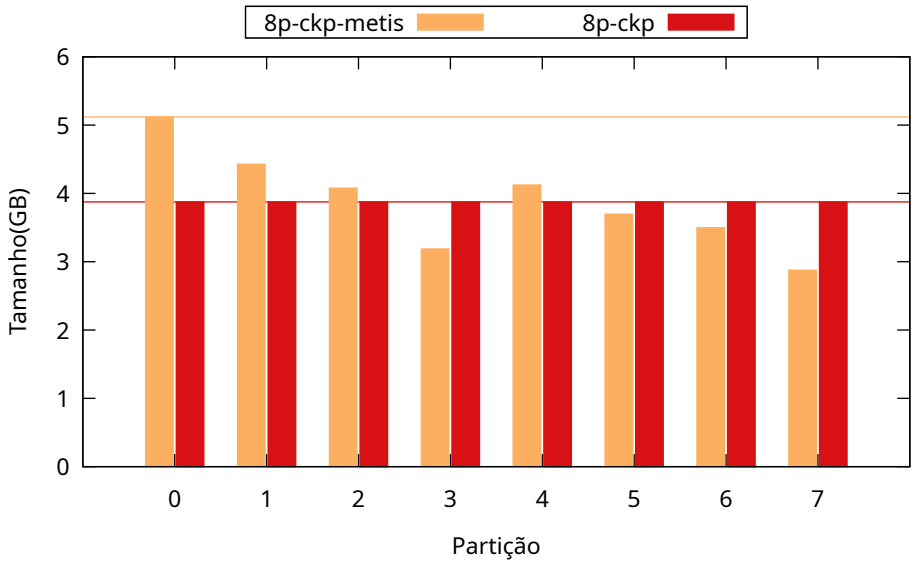


Figura 26 – Tamanhos de checkpoint por partição YCSB-A

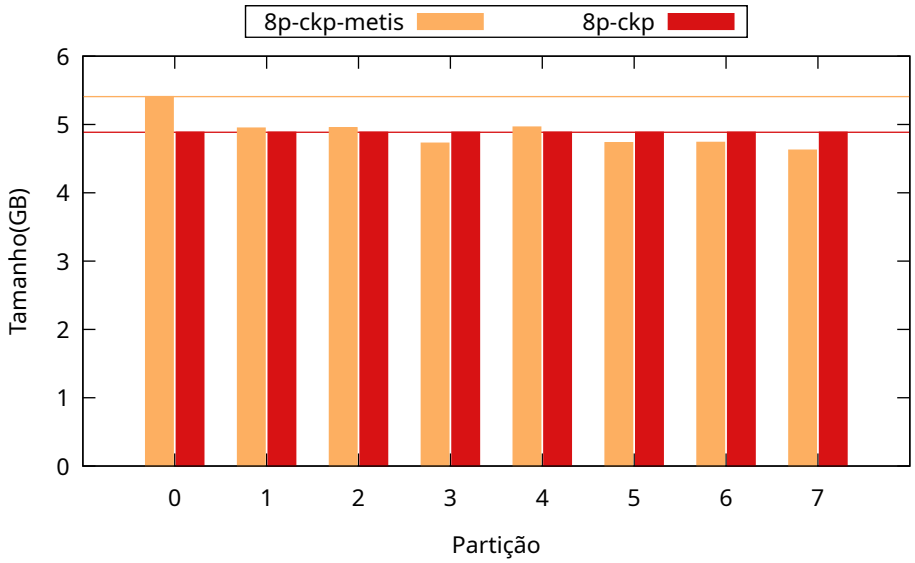


Figura 27 – Tamanhos de checkpoint por partição YCSB-D

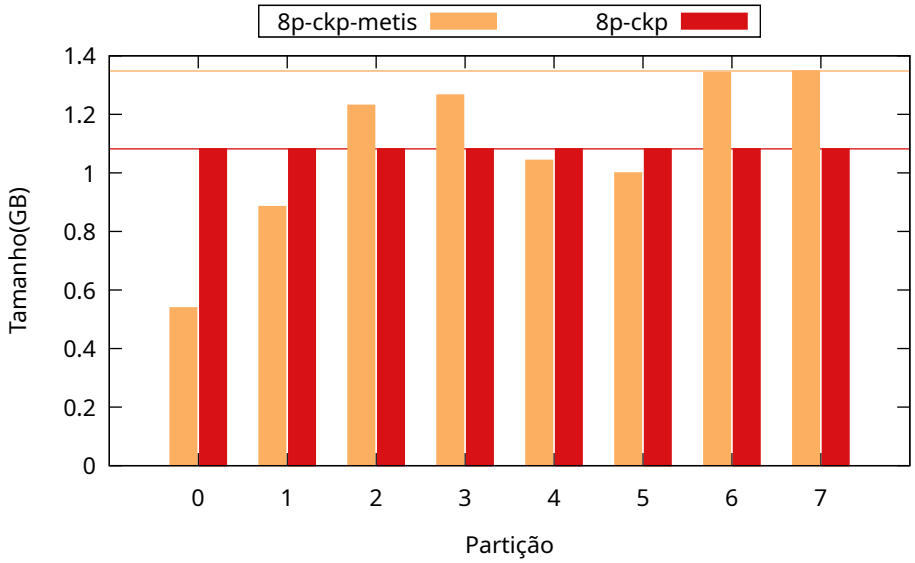


Figura 28 – Tamanhos de checkpoint por partição YCSB-E

Para aprimorar ainda mais a distribuição de estado para as *threads* de *checkpoint*, o algoritmo de reparticionamento deveria levar em conta o tamanho do estado e não apenas o número de solicitações e custos de sincronização, conforme implementado nesta abordagem. Em trabalho futuros é mencionado uma estratégia mais robusta que busca resolver este problema. A fim de mitigar o impacto causado pelo número de acessos à uma variável que poderia levar a um desbalanceamento de tamanho de partição, tentou-se aplicar uma função de log sobre o número de acessos, porém, dado o custo de processamento desta operação, ela se mostrou inviável e muito penosa ao desempenho do sistema.

O tempo para concluir o *checkpoint* está diretamente relacionado ao tamanho do maior estado da partição. A Figura 29 exibe o tempo médio necessário para completar

o checkpoint para as três cargas de trabalho (YCSB-A, YCSB-D e YCSB-E). A comparação entre os tempos do procedimento de *checkpoint*, seja com (*8p-ckp-metis*) ou sem reparticionamento (*8p-ckp*), evidencia que ambos são próximos.

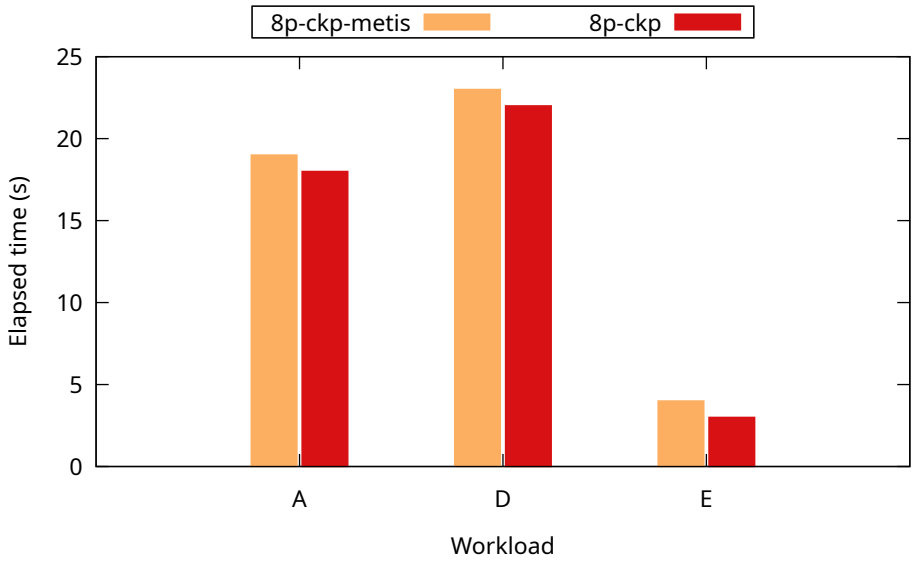


Figura 29 – Duração de checkpoint para YCSB-A, D, e E

A Figura 30 detalha o tempo gasto pelo algoritmo de reparticionamento e o necessário para o processo de *checkpoint*, que engloba a leitura do mapa em memória e sua gravação em um armazenamento estável. Uma observação relevante é o alto custo de E/S ao gravar o *checkpoint*, evidenciado pelos consideráveis tamanhos dos arquivos de *checkpoint*, conforme ilustrado nas Figuras 26, 27 e 28. Adicionalmente, a figura valida a percepção de que é viável executar um algoritmo de reparticionamento em um intervalo de tempo aceitável, especialmente quando o processador não está sendo utilizado em sua totalidade pois está esperando por operações de E/S. Embora a figura se refira à carga de trabalho YCSB-A, comportamentos similares são observados para YCSB-D e YCSB-E.

Nos experimentos, percebeu-se que a duração dos *checkpoints* está ligada ao tamanho dos pares chave-valor em memória. Testes com chave de 4 bytes e valor de 4kB resultaram em arquivos de 32GB e partições de 4GB. Em teste subsequente com valores de 1kB, buscou-se entender as variações nos tempos de *checkpoint* e reparticionamento. Com essa alteração, os arquivos de *checkpoint* reduziram, diminuindo o tempo de leitura e gravação. Notavelmente, como na Figura 31, o reparticionamento ocasionalmente superou o tempo do *checkpoint*, como no instante 24, com uma diferença de um segundo. Este achado é significativo para implementação de algoritmos de reparticionamento, destacando que o impacto está relacionado ao tamanho dos estados em memória, determinado pelo número de entradas e tamanho dos valores

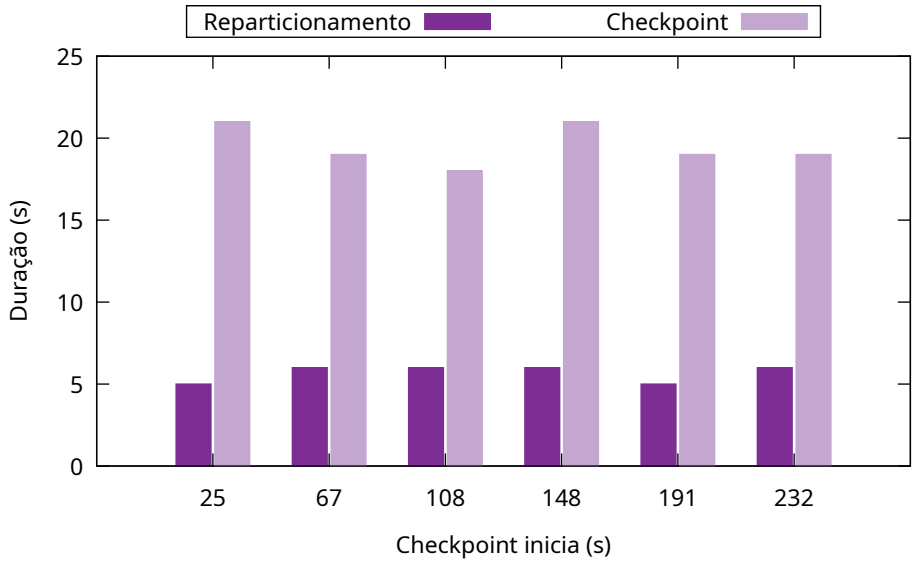


Figura 30 – Tempo *checkpoint* - YCSB-A com valores de 4kB.

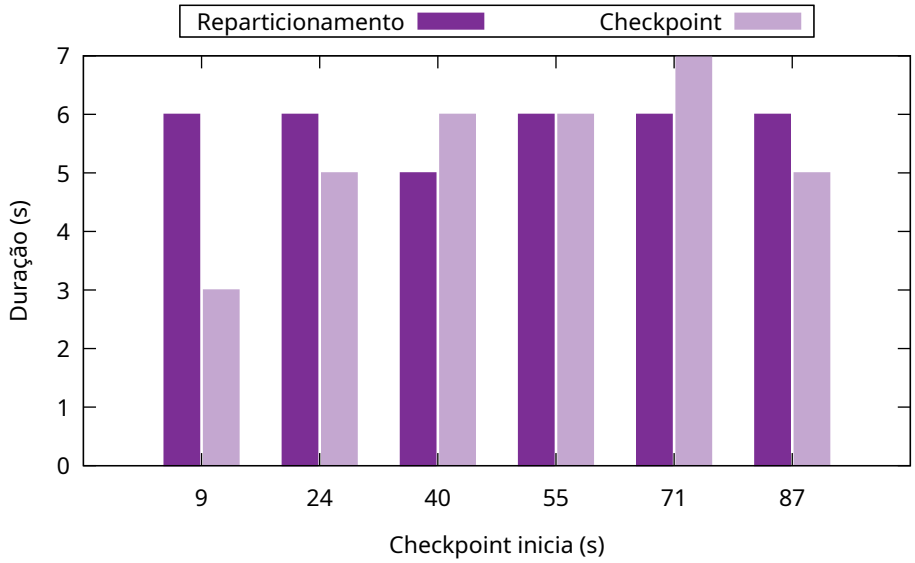


Figura 31 – Tempo *checkpoint* - YCSB-A com valores de 1kB.

5.2.6 *Makespan*

Nesta seção, concentra-se na análise do impacto de diferentes configurações de *checkpoint* sobre o *makespan*, métrica que quantifica o tempo necessário para a conclusão de uma execução. Para essa avaliação, todas as configurações experimentais foram uniformizadas com um número idêntico de solicitações, sendo os resultados detalhadamente apresentados na Figura 32, com base na carga de trabalho YCSB-A.

A partir dos dados obtidos, observa-se que a configuração *8p-ckp-metis* exibe um *makespan* similar à outra estratégia particionada que não possuí reparticionamento. Mesmo que a Figura 19 indique vazão comparáveis entre as configurações *8p-ckp-metis* e *8p-ckp*, e a Figura 26 evidencie variações nos tamanhos de partição ao adotar um particionamento variável, a eficaz distribuição de requisições entre os *threads* neu-

traliza tais discrepâncias, refletindo em um desempenho melhorado. Nota-se, assim, na Figura 32, a similaridade dos *makespans* entre *8p-ckp-metis* e *8p-ckp*.

Em suma, os resultados apontam para uma superioridade do *checkpoint* particionado em relação ao tempo de conclusão quando comparado ao método não particionado, influenciando decisivamente no tempo global de execução do programa.

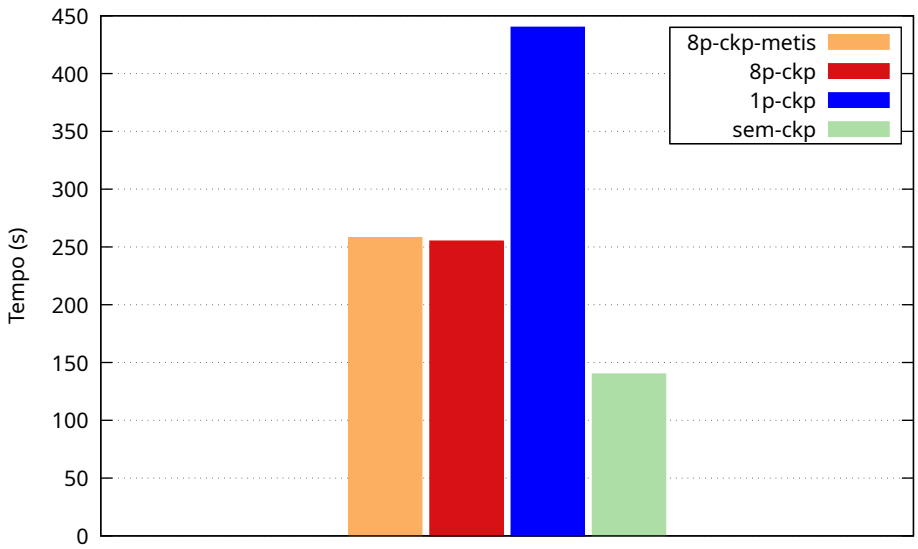


Figura 32 – Makespan - YCSB-A

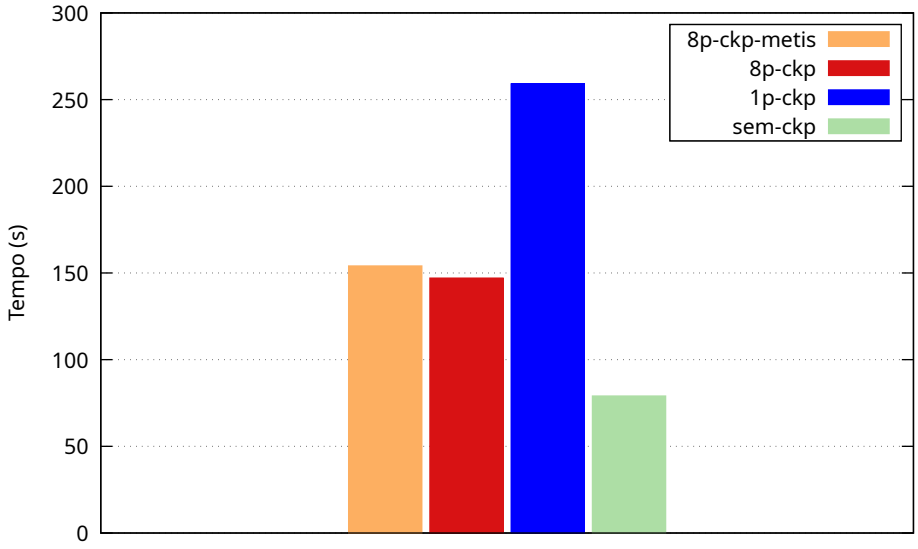


Figura 33 – Makespan - YCSB-D

Em relação à carga de trabalho YCSB-D, conforme ilustrado na Figura 33, nota-se um rendimento abaixo do esperado no *makespan*. Esta carga, como detalhado na Figura 24, representa um desafio para algoritmos de reparticionamento. Devido a este desequilíbrio, uma partição específica demora mais para ser processada pela sua *thread*, levando outras *threads* de *checkpoint* a aguardarem inativamente. Esse

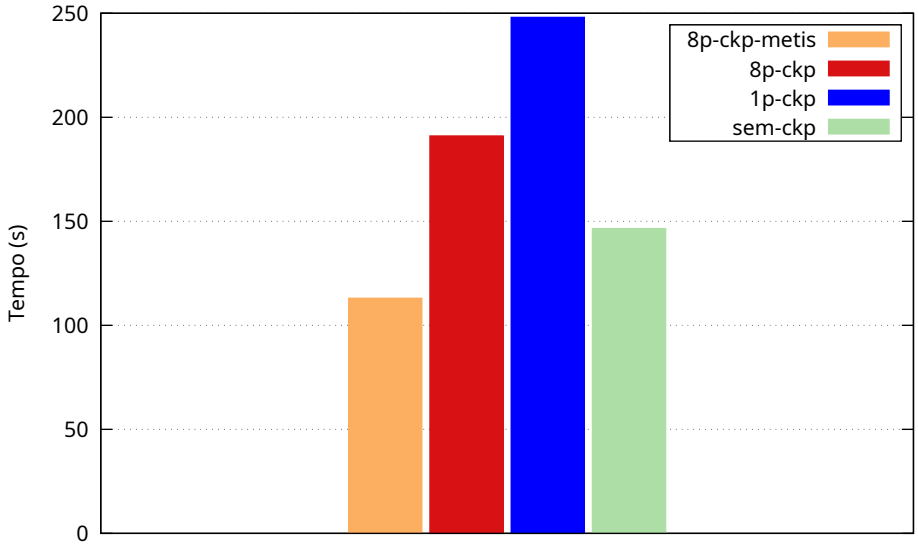


Figura 34 – Makespan - YCSB-E

cenário evidencia que, quando a *thread* encarregada tem uma quantidade maior de dados para processar, partições não balanceadas podem comprometer a eficiência de *checkpoints* realizados em paralelo. Em trabalhos futuros, é sugerido uma estratégia para mitigar esse problema.

Na Figura 34, observa-se uma melhoria notável no *makespan* com a implementação do algoritmo de reparticionamento em conjunto com o *checkpoint* para YCSB-E. Tal melhoria é atribuída à diminuição nas operações de sincronização, resultando em uma conclusão mais acelerada do processo. Os benefícios de vazão são evidenciados na Figura 21. Considerando os custos de *checkpoint* e reparticionamento, o método proposto demonstra um *makespan* mais eficaz comparado a um cenário sem a utilização de *checkpoints*.

6 CONCLUSÃO

A tecnologia avançou a ponto de permitir o surgimento do *checkpoint* paralelo, uma ferramenta capaz de aproveitar ao máximo o hardware moderno. Ao permitir que *threads* salvem partições de estado de forma simultânea, viu-se uma redução notável no tempo necessário para o *checkpoint*. Em sistemas equipados com múltiplas unidades de armazenamento persistente, essa técnica revelou-se ainda mais benéfica, pois a E/S paralela potencializou as taxas de transferência ao criar *fotos* de estados.

A implementação de reparticionamento durante o processamento de *checkpoint* proposta nesta dissertação, aprimora a gestão das partições de estado, adaptando-se dinamicamente às variações na carga de trabalho. Embora à primeira vista possa parecer que tais algoritmos de repartição são exigentes em termos de recursos, na realidade, eles aproveitam de forma inteligente os períodos de inatividade. Durante os *checkpoints* e operações de E/S, esses intervalos são aproveitados para alocar recursos de processamento previamente subutilizados, com o objetivo de melhorar a reorganização do estado do programa.

Os testes de desempenho realizados com diferentes cargas realistas de trabalho, que incluíram a coleta de dados como *makespan*, vazão, tempo de *checkpoint*, distribuição de estado particionado e acesso, tiveram como objetivo compreender o impacto de realizar o reparticionamento durante o *checkpoint*. A repartição provou ser uma estratégia com considerável valor, sobretudo quando se lida com demandas de múltiplas variáveis que podem ser agrupadas numa mesma partição. Apesar dos desafios iniciais que a repartição possa aparentar, seu valor é ressaltado pela capacidade de ofuscar seus próprios custos através do mecanismo de *checkpoint*. Esse valor persiste, mesmo quando o cenário não é o ideal.

Embora haja momentos em que o algoritmo de repartição não supere técnicas tradicionais como o *round-robin*, seu desempenho se mantém competitivo. Os dados coletados sugerem um potencial de utilizar períodos de inatividade para melhorar a organização do estado do sistema. Em certos contextos, o algoritmo pode até melhorar o desempenho geral do sistema, garantindo uma distribuição mais equilibrada. Considerações futuras devem avaliar a distribuição de estado e acesso para equilibrar o tempo de *checkpoint* com os ganhos de repartição.

Esta dissertação contribui para os sistemas tolerantes a falhas de alta vazão ao destacar o potencial do reparticionamento de dados durante o processo de *checkpoint*. Através de experimentos detalhados, demonstramos a eficácia desta técnica em uma variedade de cenários, especialmente útil em sistemas que enfrentam desafios como distribuições de estado desbalanceadas, operações sincronizantes e variações na carga de trabalho. Os benefícios do reparticionamento sobre métodos convencionais, como o *round-robin*, são destacados, oferecendo uma solução que reduz a necessida-

de de sincronização e mantém uma distribuição de estados equilibrada. Os resultados deste estudo não apenas mostram a importância do tema, mas também estabelecem uma base para pesquisas futuras na área.

Durante a realização deste trabalho, foram produzidos dois artigos publicados. O primeiro artigo foi apresentado no XXIV Workshop de Testes e Tolerância a Falhas e concentrou-se em analisar estratégias de *checkpoints*, examinando sua aplicabilidade e casos de uso atuais, conforme documentado na conformereferência (GOULART; FRANCO; MENDIZABAL, 2023). O segundo artigo, "Achieving Enhanced Performance Combining Checkpointing and Dynamic State Partitioning" (GOULART; TROMBETA *et al.*, 2023), veiculado no *IEEE 35th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, investigou os impactos do reparticionamento de dados durante o processo de *checkpoint* e fundamentou as contribuições dessa dissertação.

6.1 TRABALHOS FUTUROS

Compreendendo a necessidade de otimizar o gerenciamento de dados distribuídos, este trabalho propõe uma série de abordagens futuras que se concentram em aprimorar o reparticionamento de dados. A primeira abordagem seria a implementação de uma janela deslizante para as requisições, onde apenas as últimas n requisições seriam consideradas para o reparticionamento. Esta técnica permitiria uma análise mais atualizada, refletindo melhor os padrões de acesso aos dados mais recentes. Uma implementação preliminar desta concepção foi desenvolvida no protótipo, porém, não se alcançou a conclusão dentro do escopo deste trabalho.

Além disso, outra proposta seria a inclusão do tamanho dos valores de cada chave como um critério de reparticionamento, além da frequência de acessos e da quantidade de chaves. Esta consideração proporcionaria um balanceamento mais preciso do estado dos dados, uma vez que a contagem de chaves pode ser insuficiente para representar a distribuição dos estados que são armazenados pelas partições durante o *checkpoint*.

Em cenários com grande volume de chaves, a adoção de técnicas de amostragem é viável. Esta estratégia enfocaria chaves com valores de tamanho significativo, otimizando o salvamento do *checkpoint* ao priorizar chaves de maior custo. Outra abordagem seria executar o reparticionamento apenas sob condições específicas, como em casos de desequilíbrios acentuados ou alta necessidade de sincronização entre partições. Esta metodologia visa otimizar a alocação de recursos, adaptando-se dinamicamente às exigências do sistema.

Essas abordagens indicam a evolução para sistemas de gerenciamento de dados distribuídos mais inteligentes e adaptáveis. Estes sistemas seriam capazes de ajustar-se rapidamente às variações nos padrões de acesso e às demandas dos usuários.

REFERÊNCIAS

ALCHIERI, Eduardo; DOTTI, Fernando; MARANDI, Parisa *et al.* Boosting State Machine Replication with Concurrent Execution. *In:* 2018 Eighth Latin-American Symposium on Dependable Computing (LADC). [*S.l.*: *s.n.*], 2018. P. 77–86. DOI: 10.1109/LADC.2018.00018. Citado na p. 25.

ALCHIERI, Eduardo; DOTTI, Fernando; MENDIZABAL, Odorico M *et al.* Reconfiguring parallel state machine replication. *In:* IEEE. 2017 IEEE 36th Symposium on Reliable Distributed Systems (SRDS). [*S.l.*: *s.n.*], 2017. P. 104–113. Citado nas pp. 26, 34, 35.

BARNARD, Stephen T; SIMON, Horst D. Fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems. **Concurrency: Practice and experience**, Wiley Online Library, v. 6, n. 2, p. 101–117, 1994. Citado na p. 43.

BESSANI, Alysson *et al.* On the Efficiency of Durable State Machine Replication. *In:* 2013 USENIX Annual Technical Conference (USENIX ATC 13). San Jose, CA: USENIX Association, jun. 2013. P. 169–180. Disponível em: <https://www.usenix.org/conference/atc13/technical-sessions/presentation/bessani>. Citado nas pp. 11, 12, 15, 28.

BOITO, Francieli Zanon *et al.* A Checkpoint of Research on Parallel I/O for High-Performance Computing. **ACM Computing Surveys**, v. 51, n. 2, 2018. ISSN 0360-0300. DOI: 10.1145/3152891. Citado na p. 11.

BUI, T N; JONES, C. A heuristic for reducing fill-in in sparse matrix factorization, dez. 1993. Disponível em: <https://www.osti.gov/biblio/54439>. Citado na p. 43.

BULUÇ, Aydın *et al.* Recent advances in graph partitioning. **Algorithm engineering**, Springer, p. 117–158, 2016. Citado na p. 11.

CHANDY, K Mani; RAMAMOORTHY, Chittoor V. Rollback and recovery strategies for computer programs. **IEEE Transactions on computers**, IEEE, v. 100, n. 6, p. 546–556, 1972. Citado na p. 17.

COOPER, Brian F *et al.* Benchmarking cloud serving systems with YCSB. *In:* PROCEEDINGS of the 1st ACM symposium on Cloud computing. [*S.l.*: *s.n.*], 2010. P. 143–154. Citado na p. 44.

CRIU. **CRIU**. [*S.l.*: *s.n.*], 2021. Acesso em 25 de ago de 2021. Disponível em: <https://criu.org/>. Citado na p. 24.

CURINO, Carlo *et al.* Schism: a workload-driven approach to database replication and partitioning. **Proceedings of the VLDB Endowment**, VLDB Endowment, v. 3, n. 1-2, p. 48–57, 2010. Citado na p. 29.

DUELL, Jason. The design and implementation of berkeley lab's linux checkpoint/restart, 2005. Citado na p. 23.

EGWUTUOHA, Ifeanyi P. *et al.* A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems. **Journal of Supercomputing**, v. 65, n. 3, p. 1302–1326, 2013. ISSN 09208542. DOI: 10.1007/s11227-013-0884-0. Citado nas pp. 11, 31.

ELNOZAHY, Elmootazbellah Nabil; ALVISI, Lorenzo *et al.* A survey of rollback-recovery protocols in message-passing systems. **ACM Computing Surveys (CSUR)**, ACM New York, NY, USA, v. 34, n. 3, p. 375–408, 2002. Citado nas pp. 11, 31.

ELNOZAHY, Elmootazbellah Nabil; JOHNSON, David B; ZWAENEPOEL, Willy. Measured performance of consistent checkpointing. *In:* CONF. PROCEEDINGS of the Eleventh Symposium on Reliable Distributed Systems. [*S.l.*: *s.n.*], 1992. Citado nas pp. 19, 20.

FRANK, Alvaro *et al.* Improving checkpointing intervals by considering individual job failure probabilities. *In:* IEEE. 2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS). [*S.l.*: *s.n.*], 2021. P. 299–309. Citado na p. 23.

GAREY, Michael R; JOHNSON, David S; STOCKMEYER, Larry. Some simplified NP-complete problems. *In:* PROCEEDINGS of the sixth annual ACM symposium on Theory of computing. [*S.l.*: *s.n.*], 1974. P. 47–63. Citado na p. 36.

GOULART, Henrique; FRANCO, Álvaro; MENDIZABAL, Odorico. Checkpointing Techniques in Distributed Systems: A Synopsis of Diverse Strategies Over the Last Decades. *In:* XXIV Workshop de Testes e Tolerância a Falhas. [*S.l.*: *s.n.*], 2023. DOI: 10.5753/wtf.2023.785. Citado nas pp. 31, 57.

GOULART, Henrique S.; TROMBETA, João *et al.* Achieving Enhanced Performance Combining Checkpointing and Dynamic State Partitioning. *In:* 2023 IEEE 35th International Symposium on Computer Architecture and High Performance Computing

(SBAC-PAD). [*S.l.*: *s.n.*], 2023. P. 149–159. DOI: 10.1109/SBAC-PAD59825.2023.00024. Citado nas pp. 11, 57.

JANAKIRAMAN, G; TAMIR, Yuval. Coordinated checkpointing-rollback error recovery for distributed shared memory multicomputers. *In:* IEEE. PROCEEDINGS of IEEE 13th Symposium on Reliable Distributed Systems. [*S.l.*: *s.n.*], 1994. P. 42–51. Citado na p. 17.

JUNIOR, Everaldo Gomes; ALCHIERI, Eduardo *et al.* A Time-Phased Partitioned Checkpoint Approach to Reduce State Snapshot Overhead. *In:* PROCEEDINGS of the 12th Latin-American Symposium on Dependable and Secure Computing. La Paz, Bolivia: Association for Computing Machinery, 2023. (LADC '23), p. 100–109. DOI: 10.1145/3615366.3615417. Disponível em: <https://doi.org/10.1145/3615366.3615417>. Citado nas pp. 21, 22, 27–29.

JUNIOR, Gomes; AVILA, Everaldo de. **Redução do custo da durabilidade em Replicação Máquina de Estados através de checkpoints particionados**. 2020. Diss. (Mestrado). Citado nas pp. 25, 27, 28.

KAPRITSOS, Manos *et al.* All about eve: Execute-verify replication for multi-core servers. *In:* SYMPOSIUM ON OPERATING SYSTEMS DESIGN AND IMPLEMENTATION (OSDI). [*S.l.*: *s.n.*], 2012. P. 237–250. Citado na p. 31.

KARYPIS, George; KUMAR, Vipin. A fast and high quality multilevel scheme for partitioning irregular graphs. **SIAM Journal on scientific Computing**, SIAM, v. 20, n. 1, p. 359–392, 1998. Citado nas pp. 11, 36, 42.

KERNIGHAN, Brian W; LIN, Shen. An efficient heuristic procedure for partitioning graphs. **The Bell system technical journal**, Nokia Bell Labs, v. 49, n. 2, p. 291–307, 1970. Citado na p. 43.

KNUTH, Donald Ervin. **The Stanford GraphBase: a platform for combinatorial computing**. [*S.l.*]: AcM Press New York, 1993. v. 1. Citado na p. 31.

KOTLA, Ramakrishna; DAHLIN, Michael. High throughput Byzantine fault tolerance. *In:* IEEE. INTERNATIONAL Conference on Dependable Systems and Networks, 2004. [*S.l.*: *s.n.*], 2004. P. 575–584. Citado nas pp. 25, 31.

LEE, Chang-Gyu *et al.* Write Optimization of Log-Structured Flash File System for Parallel I/O on Manycore Servers. *In:* PROCEEDINGS of the 12th ACM International

Conference on Systems and Storage. [*S.l.*: *s.n.*], 2019. P. 21–32. DOI: 10.1145/3319647.3325828. Citado na p. 11.

LI, Bijun; XU, Wenbo; KAPITZA, Rüdiger. Dynamic State Partitioning in Parallelized Byzantine Fault Tolerance. *In:* 2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W). [*S.l.*: *s.n.*], 2018. P. 158–163. DOI: 10.1109/DSN-W.2018.00056. Citado na p. 31.

LI, Kai; NAUGHTON, Jeffrey F.; PLANK, James S. Low-latency, concurrent checkpointing for parallel programs. **IEEE transactions on Parallel and Distributed Systems**, IEEE, v. 5, n. 8, p. 874–879, 1994. Citado na p. 20.

MARANDI, Parisa Jalili; BEZERRA, Carlos Eduardo; PEDONE, Fernando. Rethinking state-machine replication for parallelism. *In:* IEEE. 2014 IEEE 34th International Conference on Distributed Computing Systems. [*S.l.*: *s.n.*], 2014. P. 368–377. Citado nas pp. 25, 26.

MENDIZABAL, Odorico M; DE MOURA, Rudá ST *et al.* Efficient and deterministic scheduling for parallel state machine replication. *In:* IEEE. 2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS). [*S.l.*: *s.n.*], 2017. P. 748–757. Citado na p. 25.

MENDIZABAL, Odorico M; DOTTI, Fernando Luıs; PEDONE, Fernando. Analysis of checkpointing overhead in parallel state machine replication. *In:* PROCEEDINGS of the 31st Annual ACM Symposium on Applied Computing. [*S.l.*: *s.n.*], 2016. P. 534–537. Citado nas pp. 11, 15.

MENDIZABAL, Odorico M; JALILI MARANDI, Parisa *et al.* Checkpointing in parallel state-machine replication. *In:* SPRINGER. INTERNATIONAL Conference on Principles of Distributed Systems. [*S.l.*: *s.n.*], 2014. P. 123–138. Citado nas pp. 17, 25.

MOSTEFAOUI, Achour; RAYNAL, Michel. Efficient message logging for uncoordinated checkpointing protocols. *In:* SPRINGER. DEPENDABLE Computing—EDCC-2: Second European Dependable Computing Conference Taormina, Italy, October 2–4, 1996 Proceedings 2. [*S.l.*: *s.n.*], 1996. P. 353–364. Citado na p. 17.

NAKANO, J. *et al.* ReViveI/O: efficient handling of I/O in highly-available rollback-recovery servers. *In:* THE Twelfth International Symposium on High-Performance Computer Architecture, 2006. [*S.l.*: *s.n.*], 2006. P. 200–211. DOI: 10.1109/HPCA.2006.1598129. Citado na p. 11.

NISHIMURA, Joel; UGANDER, Johan. Restreaming graph partitioning: simple versatile algorithms for advanced balancing. *In:* PROCEEDINGS of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining. [*S.l.*: *s.n.*], 2013. P. 1106–1114. Citado na p. 36.

PLANK, James S *et al.* **Libckpt: Transparent checkpointing under unix**. [*S.l.*]: Computer Science Department, 1994. Citado nas pp. 19, 24.

POTHEN, Alex; SIMON, Horst D; LIOU, Kang-Pu. Partitioning sparse matrices with eigenvectors of graphs. **SIAM journal on matrix analysis and applications**, SIAM, v. 11, n. 3, p. 430–452, 1990. Citado na p. 43.

QUAMAR, Abdul; KUMAR, K Ashwin; DESHPANDE, Amol. SWORD: scalable workload-aware data placement for transactional workloads. *In:* PROCEEDINGS of the 16th International Conference on Extending Database Technology. [*S.l.*: *s.n.*], 2013. P. 430–441. Citado nas pp. 11, 30.

SANDERS, Peter; SCHULZ, Christian. Think locally, act globally: Highly balanced graph partitioning. *In:* SPRINGER. INTERNATIONAL Symposium on Experimental Algorithms. [*S.l.*: *s.n.*], 2013. P. 164–175. Citado na p. 36.

TAMIR, Yuval; SEQUIN, Carlo H. Error recovery in multicomputers using global checkpoints. *In:* 13TH International Conference on Parallel Processing. [*S.l.*: *s.n.*], 1984. P. 32–41. Citado na p. 17.

TIWARI, Devesh; GUPTA, Saurabh; VAZHKUDAI, Sudharshan S. Lazy checkpointing: Exploiting temporal locality in failures to mitigate checkpointing overheads on extreme-scale systems. *In:* IEEE. 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks. [*S.l.*: *s.n.*], 2014. P. 25–36. Citado na p. 23.

TRIFUNOVIĆ, Aleksandar; KNOTTENBELT, William J. Parallel multilevel algorithms for hypergraph partitioning. **Journal of Parallel and Distributed Computing**, Elsevier, v. 68, n. 5, p. 563–581, 2008. Citado na p. 11.

TROMBETA, João Gabriel *et al.* Avaliação do desempenho do particionamento de estado em Replicação Máquina de Estados Paralela. Florianópolis, SC., 2021. Citado nas pp. 32, 33, 41.

TROMBETA, João Gabriel; MENDIZABAL, Odorico Machado. Proposta para Reparticionamento de Estado em Replicação Máquina de Estado Paralela. **Anais do Computer on the Beach**, v. 11, n. 1, p. 071–073, 2020. Citado nas pp. 11, 40.

TSOURAKAKIS, Charalampos *et al.* Fennel: Streaming graph partitioning for massive scale graphs. *In:* PROCEEDINGS of the 7th ACM international conference on Web search and data mining. [*S.l.*: *s.n.*], 2014. P. 333–342. Citado na p. 36.

ZHENG, Wenting *et al.* Fast databases with fast durability and recovery through multicore parallelism. *In:* 11TH {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14). [*S.l.*: *s.n.*], 2014. P. 465–477. Citado nas pp. 15, 22.

ZHONG, Hua; NIEH, Jason. CRAK: Linux checkpoint/restart as a kernel module. CUCS-019-01, 2001. Columbia: Department of Computer Science, Columbia University, Technical Report CUCS-019-01. Citado na p. 23.